Université de Limoges

Master 2 Cryptis - Computer Science
Faculty of Science and Technology

Cryptographic Software Development

Cryptographic Calculator

Supervising teacher :
Prof. Christophe CLAVIER

Authors :
Mohamed Lamine NDIONGUE
Thi Ha Trang NGUYEN
Minh Luan NGUYEN
Ho Nguyen PHAM
Xuan Bach TRAN

Academic year 2024 - 2025

Contents

List of Algorithms

List of Figures

Introduction

In this project, we implemented various cryptography primitives and standards in the C programming language, then we wrapped the C library into Python for GUI programming with the help of Cython.

Our work also includes test cases for each module along with a tutorial on how to build and test the library as well as to use the GUI program.

The implemented modules cover different categories, including secret key cryptography, public key cryptography, hash functions, etc.

The full list of implemented modules is specified in the following table.

Category	**Functionality**	**Variants**	**Security Level**
Basic Primitives	Pseudo Random Generator		
	Prime Tests & Generators	GMP, Fermat, Miller Rabin	
	Conversion Tools		
Hash Functions	MD5		L0
	SHA1		L0
	SHA2	SHA224, SHA256, SHA384, SHA512	L0, L1, L2, L3
	SHA3	SHA3-224, SHA3-256, SHA3-384, SHA3-512	L0, L1, L2, L3
	SHAKE	SHAKE128, SHAKE256	L1, L3
Secret Key Cryptography	DES		L0
	AES	ECB, CBC, CFB, OFB, CTR	L1, L2, L3
	HMAC		L0, L1, L2, L3
	CBC-MAC		L1, L2, L3
Public Key Cryptography	RSA Encryption	Textbook, PKCS#1-v1.5, OAEP	L0, L1, L2, L3
	RSA Signature	Textbook, PKCS#1-v1.5, PSS	L0, L1, L2, L3
	ElGamal Encryption		L0, L1
	ElGamal Signature		L0, L1
	DSA		L0, L1, L2, L3
	Diffie-Hellman Key Exchange		L0, L1
Elliptic Curve Cryptography	Elliptic Curves	Short Weierstrass, Montgomery, Twisted Edwards	L0, L1, L2, L3
	EC-ElGamal Encryption		L0, L1, L2, L3
	EC-ElGamal Signature		L0, L1, L2, L3
	ECDSA		L0, L1, L2, L3
	ECDH		L0, L1, L2, L3

Table 1: Implemented modules

The security level indicates the number of security bits corresponding to the implemented algorithms and parameters.

Security Level	L0	L1	L2	L3
Security Bits	≤ 112	128	192	256

Table 2: Supported levels of security

The next sections of this report cover all the results of our work for this project.

- *Section 2 - 6*: Introducing the cryptography primitives and explaining the implemented algorithms.
- *Section 7 - 8*: Presenting the GUI program and the integration between Python GUI and C library.
- *Conclusion*: Concluding on the result and reporting members' contribution.
- *Appendices*: Providing more details on the implementation.

1 Dependencies

1.1 GMP Library

As using the GMP library is a requirement of this project, we used the library in the development of the C library for the essential functionalities, including but not limited to:

- Storing integers with `mpz_t` and related functions.
- Generate random numbers and primes with `gmp_randstate_t`, `mpz_urandomb`, and `mpz_urandomm`.
- Conversion between integers and bytes with `mpz_import` and `mpz_export` functions.
- Modular arithmetic on big numbers, e.g. `mpz_invert`, `mpz_mod`, `mpz_powm`, etc.

1.2 Cython

For this project, we implemented the Graphical User Interface (GUI) in Python with the PyQt framework. Thus, the C library needs to be callable from the Python program. We decided to wrap modules of the C library into a Python library with Cython [BBC+11]. The detailed process of wrapping C modules into Python modules will be presented in the later section 7.

2 Basic Primitives

The implementation of the many cryptosystems in this project shares the requirement for basic cryptography primitives, such as a PRNG, large prime generator, conversion functions, etc. Therefore, these primitives shall be implemented from the start and described before we explore other cryptography functionalities.

2.1 PRNG

As mentioned above, we utilized the random generator functions from the GMP library to implement our PRNG - Pseudo Random Number Generator.

```c
unsigned int get_random_seed();
void rng_init(gmp_randstate_t state);
void rng_init_with_seed(gmp_randstate_t state, unsigned int seed);
void rand_int_b(mpz_t result, gmp_randstate_t state, unsigned long bits);
void rand_int_m(mpz_t result, gmp_randstate_t state, const mpz_t n);
void rand_bytes(char *buf, gmp_randstate_t state, unsigned int byte_len);
```

Secure Randomness

When users specify no seed, the program is expected to use a random seed from a cryptographically secure random source. In particular, we utilized two sources of secure randomness:

- For Linux/MacOS environment: seeds are read from `/dev/urandom`.
- For Windows environments: seeds are returned from the `RtlGenRandom()` function.

We noted that users who want to use specific seeds are expected to know what they are doing.

```c
#define RAND_SOURCE "/dev/urandom"
unsigned int get_random_seed() {
    unsigned long int rand_src;
#ifdef WINDOWS_H
    while (!RtlGenRandom(&rand_src, sizeof(unsigned int))){};
#else
    FILE *fp = fopen(RAND_SOURCE, "rb");
    fread(&rand_src, sizeof(unsigned int), 1, fp);
    fclose(fp);
#endif
    return rand_src;
}
```

2.2 Primality Test

We implemented three probabilistic tests for primality in our library: the GMP test, the Fermat test, and the Miller-Rabin test.

```c
typedef enum prime_test {
    GMP_TEST,
    FERMAT_TEST,
    MILLER_RABIN_TEST
} prime_test_t;

int primality_test(mpz_t n, gmp_randstate_t state, int t, prime_test_t test);
```

The argument `t` provides a trade-off between performance and the test result's certainty level. These tests are later used in the prime generators.

Algorithm 1 Fermat Primality Test

Input: Integer n, Random state state, Number of iterations t
Output: **Prime** if n is likely prime, otherwise **Composite**

repeat
  Generate random integer a in range $[2, n-2]$
  Compute $a^{n-1} \mod n$
  if $a^{n-1} \not\equiv 1 \mod n$ **then**
    return Composite
  end if
  $t \leftarrow t - 1$
until $t = 0$
return Prime

Algorithm 2 Miller-Rabin Primality Test

Input: Integer n, Random state state, Number of iterations t
Output: **Prime** if n is likely prime, otherwise **Composite**

if n is even **then**
  return Composite
end if
Compute $n - 1 = 2^s \cdot r$ where r is odd
repeat
  Generate random integer a in range $[2, n-2]$
  Compute $y \leftarrow a^r \mod n$
  if $y \not\equiv 1 \mod n$ and $y \not\equiv -1 \mod n$ **then**
    for $j = 1$ to $s - 1$ **do**
      $y \leftarrow y^2 \mod n$
      if $y \equiv 1 \mod n$ **then**
        return Composite
      end if
      if $y \equiv -1 \mod n$ **then**
        Break
      end if
    end for
    if $y \not\equiv -1 \mod n$ **then**
      return Composite
    end if
  end if
  $t \leftarrow t - 1$
until $t = 0$
return Prime

2.3 Prime Generator

Our implementation of prime generator allows the selection of one of three primality tests described above.

```
void gen_prime_b(mpz_t n, gmp_randstate_t state, int b, int k, int t, prime_test_t test);
void gen_prime_m(mpz_t n, gmp_randstate_t state, mpz_t m, int k, int t, prime_test_t test);
```

We allowed the preparation of small primes to optimize the algorithm, but this also means that the generated prime cannot be smaller than the prepared numbers.

Algorithm 3 Prime Generation by Size in Bits

Input: Bit size b, Random state state, Small primes count k, Iterations t, Primality test test
Output: Prime number n

repeat
  Generate random odd integer q of $b-1$ bits
  Set most significant bit to ensure q has b bits
  if q is even **then**
    $q \leftarrow q+1$
  end if
until q is not divisible by small primes
repeat
  if q is divisible by any small prime **then**
    $q \leftarrow q+2$
    Restart checking
  else if q fails primality test **then**
    $q \leftarrow q+2$
  end if
until q is prime
return q

Algorithm 4 Prime Generation by Modulo

Input: Modulo bound m, Random state state, Small primes count k, Iterations t, Primality test test
Output: Prime number n

repeat
  Generate random integer q in range $[2, m]$
  if $q = 2$ **then**
    return q
  end if
  if q is even **then**
    $q \leftarrow q+1$
  end if
until q is not divisible by small primes
repeat
  if $q \geq m$ **then**
    Exit with failure
  end if
  if q is divisible by any small prime **then**
    $q \leftarrow q+2$
    Restart checking
  else if q fails primality test **then**
    $q \leftarrow q+2$
  end if
until q is prime
return q

2.4 Conversion Utilities

There are several functions providing conversion utilities implemented for usage in other cryptography primitives or in test scripts.

```
typedef enum byte_order
{
    NATIVE,
    BIG,
    LITTLE
} byte_order_t;

void bytes_to_bigint(mpz_t result, const unsigned char *buf, size_t len, byte_order_t order);
void bigint_to_bytes(unsigned char *buf, size_t *len, const mpz_t num, byte_order_t order);
void hex_to_bigint(mpz_t result, const char *buf);
void bigint_to_hex(char **buf, const mpz_t num);
void string_to_hex(unsigned char *output, const char *input, size_t input_len);
void hex_to_string(char *output, const unsigned char *input, size_t input_len);
size_t count_bytes(const mpz_t n);
unsigned char *pkcs7_padding(unsigned char *input, size_t len, size_t *out_len, size_t block_size);
unsigned char *pkcs7_unpadding(unsigned char *input, size_t len, size_t *out_len, size_t block_size);
```

The PKCS#7 padding is a standard padding scheme, see more details in H.

3 Hash Functions

A hash function, in cryptography, is a mapping that from an arbitrary input data gives a fixed-size output.

$$h : \mathbb{F}_2^* \longrightarrow \mathbb{F}_2^l. \text{ With } l \text{ a fixed length.}$$

It is a type of function that is difficult to invert. That is, the image by a hash function is calculated easily and very efficiently, but calculating its inverse seems difficult or even practically impossible. Thus, such a function is said to be one-way. It is defined according to three main properties:

- Preimage resistance: that is, for any hash value h, it would be difficult to find a message m such that $h(m) = h$. This thus defines the notion of a one-way function, and those that lack this property are deemed vulnerable to preimage attacks.
- Second preimage resistance: for any message m_1, it is generally difficult to find a message $m_2 \neq m_1$ such that $h(m_1) = h(m_2)$ and the attack related to this property is called second preimage attack.
- Collision resistance: this property is defined from an angle where it is difficult to find two messages m_1 and m_2 such that $h(m_1) = h(m_2)$. To achieve this property, a hash value at least twice as long as that required for preimage resistance is necessary. If the key is not long enough, a collision can be found by a birthday attack.

We distinguish different hash functions such as: MD5, SHA1, SHA2, and SHA3.

3.1 MD5

3.1.1 Background

Invented in 1996 by Ronald Rivest, MD5 (Message Digest) is a cryptographic hash function that from an input (a file, password, or text message) calculates its digital fingerprint (specifically a sequence of 128 bits = 16x8 or 32 hexadecimal characters) with a high probability that two different inputs will produce two different fingerprints.

3.1.2 Overall structure

```c
#ifndef MD5_H
#define MD5_H

#include "common.h"

#define MD5_BLOCK_SIZE 16 // MD5 outputs a 16 byte digest
#define MD5_DIGEST_SIZE 16

typedef struct{
    BYTE data[64];
    uint32_t datalen;
    uint64_t bitlen;
    uint32_t state[4];
} md5_ctx;

void md5_init(md5_ctx *ctx);
void md5_update(md5_ctx *ctx, const BYTE data[], size_t len);
void md5_final(md5_ctx *ctx, BYTE hash[]);

void *md5(const void *m, size_t len, void *md, size_t md_len);

#endif
```

Explanation of the MD5 header file

- `#ifndef MD5_H`: This line of code instructs the processor to check if the `MD5_H` symbol has not yet been defined, and if so, the code between `#ifndef` and `#endif` will be included.

- `#define MD5_H`: while this line shows that even if the `MD5_H` symbol exists, it defines it. That is, the next time the preprocessor encounters `#ifndef MD5_H`, it will not skip the code block between `#ifndef` and `#endif`, because `MD5_H` will then be already defined. So this part allows defining constants that are valid throughout the program.

- `#include "common.h"`: this instruction is a preprocessor directive in C/C++ that tells the compiler to include the contents of the header file named `common.h` at this precise location.
- `#define MD5_BLOCK_SIZE 16` and `#define MD5_DIGEST_SIZE 16`: In this part, we define constants that are valid throughout the program and are invariable.
    - `#define MD5_BLOCK_SIZE 16`: sets the value of `MD5_BLOCK_SIZE` to 16, and this constant symbolizes the block size used by MD5, which processes data in 512-bit (64 bytes) blocks, and the output is a fixed length of 128 bits.
    - `#define MD5_DIGEST_SIZE 16`: this line attributes `MD5_DIGEST_SIZE` a constant value of 16 (16 bytes <=> 128 bits) which remains valid and unchanged throughout the program.
- The `typedef struct` sequence defines a structure attributed to `md5_ctx`, which is actually a type with which we define various variables. This sequence consists of:
    - `BYTE data[64]`: `BYTE` is a type previously defined in the `common.h` file, replacing `unsigned char`, and data is an array of size 64.
    - `uint32_t datalen`: `uint32_t` is a 32-bit unsigned integer data type defined to be used when the size of variables needs to be precisely controlled, such as in embedded programming or network communication protocols.
    - `uint64_t bitlen`: Same definition as the previous one, except that this one is 64 bits.
    - `uint32_t state[4]`: `state` is a variable of size 4.

Prototypes: In this section, we define the prototypes of the different functions used for the **MD5** hash function. Among these, we have:

- **void md5_init(md5_ctx *ctx)**: This is an initialization function for the MD5 context to calculate an MD5 hash consisting of an argument and not returning any value.
- **void md5_update(md5_ctx *ctx, const BYTE data[], size_t len)**: It is a function to update the MD5 context with additional data, consisting of three arguments.
- **void md5_final(md5_ctx *ctx, BYTE hash[])**: It is a function consisting of two arguments that ends the MD5 hash calculation and produces the final result in a byte array.
- **void *md5(const void *m, size_t len, void *md, size_t md_len)**: It is a generator function defined to call all the previous functions.

3.1.3 Usage example

A small change in the hashing message can significantly change its fingerprint.

Example 1. MD5("University of Limoges") = 45a9a2c0d686b8a1a8e505491db76db3.

Example 2. MD5("university of Limoges") = a95a3eb6ed4cfe0782519130f7c80f4b.

Despite the importance of this hash function, since 2004 it has been considered cryptographically insecure due to the discovery of complete collisions made by a Chinese team composed of Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Their method allowed them to discover a complete collision where two different messages produce the same fingerprint without using an exhaustive search method.

During our project, we developed this hash function to include it in our calculator software developed by our team. The detailed explanation of the implementation can be found in the appendix B.

3.2 SHA1

3.2.1 Background

Successor to SHA0, which was completely phased out by NIST, SHA1 is a Secure Hash Algorithm designed by the NSA (National Security Agency) of the United States and then published as an information processing standard by NIST. It results in a hashed or "condensed" value of 160 bits (20 bytes) represented in a 40-character hexadecimal format.

3.2.2 Overall structure

```c
#define SHA1_BLOCK_SIZE 20 // SHA1 outputs a 20 byte digest
#define SHA1_DIGEST_SIZE 20

typedef struct
{
        BYTE data[64];
        uint32_t datalen;
        uint64_t bitlen;
        uint32_t state[5];
        uint32_t k[4];
} sha1_ctx;

/********************** FUNCTION DECLARATIONS **********************/
void sha1_init(sha1_ctx *ctx);
void sha1_update(sha1_ctx *ctx, const BYTE data[], size_t len);
void sha1_final(sha1_ctx *ctx, BYTE hash[SHA1_DIGEST_SIZE]);
void *sha1(const void *m, size_t len, void *md, size_t md_len);
```

In the header file, we have the definition of the macros #define SHA1_BLOCK_SIZE and #define SHA1_DIGEST_SIZE, each set to 20 bytes in size. Then, we define a structure type sha1_ctx, which includes BYTE data[64], uint32_t datalen, uint64_t bitlen, uint32_t state[5], and uint32_t k[4], with BYTE being an unsigned char type of 8 bits defined in the common.h file. Afterward, we declare the prototypes of the functions sha1_init(), sha1_update(), sha1_final(), and *sha1(), all of which are of type void, meaning they do not return any value.

In the implementation file, we first include the previously defined sha1.h library, followed by the definition of a macro ROTLEFT(a, b) ((a $\ll$ b) | (a $\gg$ (32 - b))) previously defined in MD5.

```c
#include "sha1.h"

/**************************** MACROS ******************************/
#define ROTLEFT(a, b) ((a << b) | (a >> (32 - b)))
```

The main functionality of `SHA-1` is executed with a triple `(sha1_init, sha1_update, sha1_final)`. The detailed explanation for the implementation of these functions as well as the essential `sha1_transform` function can be found in the appendix C.

3.3 SHA2 Family

3.3.1 Background

Cryptanalyzed in 2004 (by a Chinese team) and in 2005 (by Xiaoyun Wang and his co-authors Yiqun Lisa Yin and Hongbo Yu), the MD5 and SHA1 algorithms are no longer considered secure due to proven security shortcomings from successful attacks. Therefore, NIST and other international standards organizations advocate replacing these two previous hash algorithms with the SHA2 algorithm or the SHA3 algorithm, which have not yet experienced attacks proving their security flaws.

SHA2, defined as a family of hash functions designed by the NSA (National Security Agency), includes the hash function variants SHA-256, SHA-224, SHA-512, SHA-384, SHA-512/256, and SHA-512/224, which are recent truncated versions of SHA-512. Thus, there are two types of hash functions for SHA2: SHA-256 and SHA-512, while the others are variants of one or the other. These two functions (SHA-256 and SHA-512) have the same structure but differ in the size of the words and blocks used.

Like any hash function, SHA2 functions take an input message of arbitrary size and produce a fixed-size output, with the size of the hash indicated by the suffix (224 bits for SHA-224, 256 bits for SHA-256, 384 bits for SHA-384, and 512 bits for SHA-512).

3.3.2 Overall structure

The constant values are defined for different levels of security of SHA-2 functions.

#define SHA224_DIGEST_SIZE (224 / 8): The constant SHA224_DIGEST_SIZE represents the 28-byte result obtained from the 224-bit size of SHA-224 divided by 8.

#define SHA256_DIGEST_SIZE (256 / 8): The constant SHA256_DIGEST_SIZE represents the 32-byte result obtained from the 256-bit size of SHA-256 divided by 8.

#define SHA384_DIGEST_SIZE (384 / 8): The constant SHA384_DIGEST_SIZE represents the 48-byte result obtained from the 384-bit size of SHA-384 divided by 8.

#define SHA512_DIGEST_SIZE (512 / 8): The constant SHA512_DIGEST_SIZE represents the 64-byte result obtained from the 512-bit size of SHA-512 divided by 8.

#define SHA256_BLOCK_SIZE (512 / 8): The constant SHA256_DIGEST_SIZE represents the 64-byte result obtained from the 512-bit size of SHA-256 divided by 8.

#define SHA512_BLOCK_SIZE (1024 / 8): The constant SHA512_DIGEST_SIZE represents the 128-byte result obtained from the 1024-bit size of SHA-512 divided by 8.

#define SHA384_BLOCK_SIZE SHA512_BLOCK_SIZE: Since SHA-384 uses the same block structure as SHA-512, the size is the same, 128 bytes.

#define SHA224_BLOCK_SIZE SHA256_BLOCK_SIZE: Since SHA-224 uses the same block structure as SHA-256, the size is the same, 64 bytes.

N.B.: The above code defines constants for the output sizes and block sizes for the SHA-224, SHA-256, SHA-384, and SHA-512 algorithms by converting the bit sizes to bytes.

```c
#ifndef SHA2_TYPES
#define SHA2_TYPES
typedef unsigned char uint8;
typedef unsigned int uint32;
typedef uint64_t uint64;
#endif

typedef struct
{
    uint64 tot_len;
    uint64 len;
    uint8 block[2 * SHA256_BLOCK_SIZE];
    uint32 h[8];
} sha256_ctx;

typedef struct
{
    uint64 tot_len;
    uint64 len;
    uint8 block[2 * SHA512_BLOCK_SIZE];
    uint64 h[8];
} sha512_ctx;

typedef sha512_ctx sha384_ctx;
typedef sha256_ctx sha224_ctx;
```

In this code, we first have the definition of the types **uint8** (typedef unsigned char uint8) to represent an unsigned 8-bit character, **uint32** (typedef unsigned int uint32) to represent an unsigned 32-bit integer, and **uint64** (typedef uint64_t uint64) is a standard type defined in the <stdint.h> library that represents an unsigned 64-bit integer.

Next, the definition of two context structures for SHA-256 and SHA-512, which are the main hash functions, and the others (SHA-224 with the type defined as typedef sha256_ctx sha224_ctx, and SHA-384 with the type defined as typedef sha512_ctx sha384_ctx) are variants.

Finally, the definition of the prototypes of the different SHA2 functions: their initialization function, their

update function, their final calculation function for each, and the provision of the digest and their function, which performs the complete calculation of each type of SHA-λ with $\lambda \in \{224, 256, 384, 512\}$.

```
void sha224_init(sha224_ctx *ctx);
void sha224_update(sha224_ctx *ctx, const uint8 *message, uint64 len);
void sha224_final(sha224_ctx *ctx, uint8 *digest);
void sha224(const uint8 *message, uint64 len, uint8 *digest);

void sha256_init(sha256_ctx *ctx);
void sha256_update(sha256_ctx *ctx, const uint8 *message, uint64 len);
void sha256_final(sha256_ctx *ctx, uint8 *digest);
void sha256(const uint8 *message, uint64 len, uint8 *digest);

void sha384_init(sha384_ctx *ctx);
void sha384_update(sha384_ctx *ctx, const uint8 *message, uint64 len);
void sha384_final(sha384_ctx *ctx, uint8 *digest);
void sha384(const uint8 *message, uint64 len, uint8 *digest);

void sha512_init(sha512_ctx *ctx);
void sha512_update(sha512_ctx *ctx, const uint8 *message, uint64 len);
void sha512_final(sha512_ctx *ctx, uint8 *digest);
void sha512(const uint8 *message, uint64 len, uint8 *digest);

void *sha2(const void *m, size_t len, void *md, size_t md_len);
```

The detailed explanation of SHA-2 implementation can be found in the appendix D.

3.3.3 Usage example

In this section, we first have the important inclusion of the **sha2.h** library, which is used to call and manipulate functions in the main function. Then, we have test functions, namely **test(const char *vector, uint8 *digest, uint32 digest_size)** and **test_sha2()**, which perform tests for SHA2 hashing. They are defined as follows:

test(const char *vector, uint8 *digest, uint32 digest_size)

This test function compares an expected test vector with the computed digest to verify the validity of the hash. It converts the binary digest into a hexadecimal string and stores it in output. Then, it compares this string with the expected vector and returns 1 if they are identical; otherwise, it returns 0.

```
#include "sha2.h"

int test(const char *vector, uint8 *digest, uint32 digest_size)
{
    char output[2 * SHA512_DIGEST_SIZE + 1];
    int i;

    output[2 * digest_size] = '\0';
    for (i = 0; i < (int)digest_size; i++)
    {
        snprintf(output + 2 * i, 3, "%02x", digest[i]);
    }

    if (strcmp(vector, output))
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

```

As for the **test_sha2** function, it verifies the validity of the SHA-224, SHA-256, SHA-384, and SHA-512 implementations using known test vectors. It defines test vectors for each hashing algorithm and three test messages of different lengths. For each algorithm, it computes the hash of the test messages and compares the

results with the expected test vectors using the test function. If all results match the test vectors, it displays "PASSED"; otherwise, it displays "FAILED". The function dynamically allocates memory for a message of 1,000,000 'a' characters and ensures that this memory is freed at the end. It uses the sha224, sha256, sha384, and sha512 functions to compute the hashes of the test messages. Finally, it displays the test results for each hashing algorithm.

The main function of the execution test file displays a header indicating the start of the SHA-2 tests. It then calls the test_sha2 function to run validation tests for the SHA-224, SHA-256, SHA-384, and SHA-512 hashing algorithms.

3.4 SHA3 Family

3.4.1 Background

SHA-3 (Secure Hash Algorithm 3) is the latest member of the Secure Hash Algorithm family of standards, released by NIST in 2015 [NIS15]. While it belongs to the same series of standards for secure hash functions, the design of SHA-3 is completely different from the Merkle–Damgård structure of SHA-1 and SHA-2. Indeed the standardized functions in the SHA-3 family are from another family of cryptography algorithms called Keccak, developed and maintained by the Keccak team [BDPA]. During the competition for the SHA-3 standardization, the team proposed the *sponge construction*, which has become a powerful cryptographic tool and a great source of inspiration for designing cryptography primitives, e.g. Poseidon hash functions, Ascon cryptosystem, etc.

3.4.2 Overall structure

At the core of Keccak is a set of seven permutations called KECCAK-f[b], with width b chosen between 25 and 1600 by multiplicative steps of 2. Depending on b, the resulting function ranges from a toy cipher to a wide function. The instances proposed for SHA-3 use exclusively KECCAK-f[1600] for all security levels. When restricted to the case $b = 1600$, the KECCAK family is denoted by KECCAK[c]; in this case, r is determined by the choice of c. In particular,

$$\text{KECCAK}[c] = \text{SPONGE}[\text{KECCAK} - p[1600, 24], \textit{pad10*1}, 1600 - c].$$

Thus, given an input bit string N and an output length d,

$$\text{KECCAK}[c](N, d) = \text{SPONGE}[\text{KECCAK} - p[1600, 24], \textit{pad10*1}, 1600 - c](N, d).$$

Function	Definition	Security Level
SHA3-224	Keccak[448]($M \parallel 01, 224$)	L0
SHA3-256	Keccak[512]($M \parallel 01, 256$)	L1
SHA3-384	Keccak[768]($M \parallel 01, 384$)	L2
SHA3-512	Keccak[1024]($M \parallel 01, 512$)	L3
SHAKE128	Keccak[256]($M \parallel 1111, d$)	L1
SHAKE256	Keccak[512]($M \parallel 1111, d$)	L3

Table 3: SHA-3 and SHAKE Functions with Security Levels

To understand how SHA-3 works internally, we need to understand two main components of KECCAK: the permutation functionality and the sponge construction.

KECCAK-f permutation. Basically, KECCAK's permutation functionality applies five different algorithms on the state multiple rounds. The detailed explanation of these algorithms can be found in appendix E.

Algorithm 5 Keccak-f Permutation Function

Input: State array $st[25]$
Output: Updated state array $st[25]$
1: **for** $r = 0$ to $Nr - 1$ **do**
2: **Apply Theta:** Call *Theta Step*
3: **Apply Rho and Pi:** Call *Rho and Pi Steps*
4: **Apply Chi:** Call *Chi Step*
5: **Apply Iota:** Call *Iota Step*
6: **end for**

Sponge construction. It is a framework for specifying functions on binary data with arbitrary output length. The construction employs the following three components:

- An underlying function on fixed-length strings, denoted by f,
- A parameter called the *rate*, denoted by r, and
- A padding rule, denoted by *pad*.

The function that the construction produces from these components is called a *sponge function*, denoted by $\text{SPONGE}[f, \text{pad}, r]$. A sponge function takes two inputs: a bit string, denoted by N, and the bit length, denoted by d, of the output string. The sponge function is evaluated as:

$$\text{SPONGE}[f, \text{pad}, r](N, d).$$

The analogy to a sponge is that an arbitrary number of input bits are *absorbed* into the state of the function, after which an arbitrary number of output bits are *squeezed* out of its state.

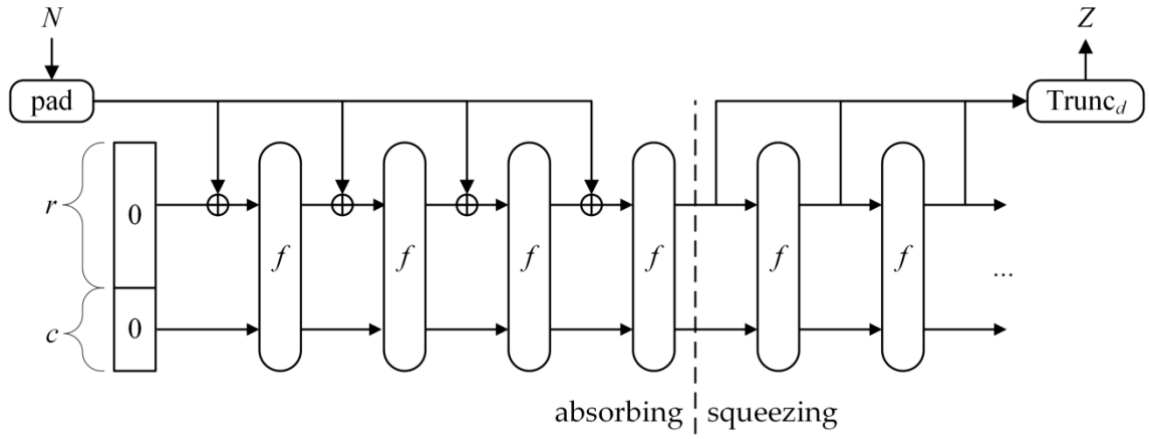


Figure 1: The sponge construction: $Z = \text{SPONGE}[f, \text{pad}, r](N, d)$[NIS15].

To implement these two main components of the SHA-3 family, we have the following header file:

```
#define L 6
#define W 64
#define B 1600
#define Nr 24

#define SHA3_224_DIGEST_SIZE 28
#define SHA3_256_DIGEST_SIZE 32
#define SHA3_384_DIGEST_SIZE 48
#define SHA3_512_DIGEST_SIZE 64

typedef struct sha_3_ctx
{
    union
    {
        uint8_t b[B / 8]; // in 8-bit bytes
```

```
        uint64_t q[25];   // in 64-bit words
    } state;
    unsigned int cnt;  // byte counter
    unsigned int rate; // in 8-bit bytes
    size_t md_len;     // in 8-bit bytes
} sha3_ctx_t;

// Keccak-f[b] permutation
void sha3_keccakf(uint64_t st[25]);

// SHA3
void sha3_init(sha3_ctx_t *ctx, size_t md_len);
void sha3_update(sha3_ctx_t *ctx, const void *m, size_t len);
void sha3_final(void *md, sha3_ctx_t *ctx);
void *sha3(const void *m, size_t len, void *md, size_t md_len);

// SHAKE
#define shake128_init(ctx) sha3_init(ctx, 16)
#define shake256_init(ctx) sha3_init(ctx, 32)
#define shake_update sha3_update

void shake_xof(sha3_ctx_t *ctx);
void *shake_out(sha3_ctx_t *ctx, void *out, size_t len);
```

The main difference between fixed-length SHA3-X hash functions and SHAKE-X extended-output hash functions is the finalization of the internal state. For fixed-length digest, the `sha3_final` is called after the hashing message is fully absorbed. Otherwise, the `shake_xof` is called by XOF functions to change from `absorbing` mode to `squezzing` mode, which would continue to update the state to return output of arbitrary length.

4 Secret Key Cryptography

4.1 DES Encryption

4.1.1 Background

DES (Data Encryption Standard) is a symmetric key algorithm for encrypting and decrypting data. Developed in the 1970s by IBM and later adopted as a federal standard in the United States, DES was widely used for securing sensitive data in various applications, including financial transactions and communications.

DES operates on 64-bit data blocks and uses a 56-bit key, which is expanded into a set of 16 subkeys, each used in one of the 16 rounds of the encryption process. DES employs a combination of permutation, substitution, and Feistel network operations to provide strong encryption.

Despite its historical significance, DES is now considered insecure for many applications due to its short key length, which makes it vulnerable to brute-force attacks. However, it remains an important algorithm for understanding the fundamentals of cryptography.

4.1.2 Overall structure

The library consists of 3 main files:

- The header file (des.h): defines the constants, data types, and functions of the DES encryption algorithm.
- The source file (des.c): provides the implementations for all of the defined functions.
- The test file (des.test.c): gives an example of how to use the DES encryption library.

The DES 56-bit secret key is transformed into 16 subkeys (the remaining 8 bits are either discarded or used as parity check bits).

Algorithm 6 DES KeySchedule

Input: Key *key*
Output: *subkeys*: Expanded subkeys

1: $key\_pc1 \leftarrow$ Permutation_PC1(key)
2: $left\_half \leftarrow$ Extract the first 28 bits of *key_pc1*
3: $right\_half \leftarrow$ Extract the last 28 bits of *key_pc1*
4: **for** $i \leftarrow 0$ **to** $DES\_ROUNDS - 1$ **do**
5: $\quad c \leftarrow left << SHIFT[i]$
6: $\quad d \leftarrow right << SHIFT[i]$
7: $\quad subkeys[i] \leftarrow$ Permutation_PC2($(c << 28) \quad OR \quad d$)
8: **end for**
9: **return** *subkeys*

The library takes in an array of hex blocks of 64 bits, the key, the number of blocks, and the mode (encryption/decryption). DES encryption and decryption are different only in the order of the subkeys. DES algorithm starts by pre-calculating the subkeys. At the beginning and the end of the process, there are the Initial Permutation and the Inverse Permutation. In between, before the main rounds, the block is divided into two 32-bit halves and processed alternately; this criss-crossing is known as the Feistel scheme.

Algorithm 7 DES

Input: Data *input*, Key *key*, Number of Block *num_block*, Encryption mode *mode*
Output: *output*: Ciphertext

```
subkeys ← DES_KeySchedule(key)
for i ← 0 to num_block − 1 do
    block ← input[i]
    block ← InitialPermutation(block)
    block ← DES_Feistel(block, mode)
    block ← InverseInitialPermutation(block)
    output[i] ← block
end for
return output
```

Each round processes the data as follows:

- Split Block: Divide the 64-bit block into two 32-bit halves ($left$ and $right$).
- F-Function (Applied to $right$)
  - Expansion: Expand R from 32 to 48 bits.
  - XOR with Subkey: Mix with the round's 48-bit subkey.
  - S-Box Substitution: Split into 8 groups of 6 bits. Each group is substituted using one of the 8 S-boxes to produce 4 bits. Total output: 32 bits.
  - Permutation: Rearrange the 32-bit result.
- XOR with $left$: The output of the F-function is XORed with $left$.
- Swap Halves: $left$ and $right$ are swapped (except in the final round).

Algorithm 8 DES Feistel

```
procedure DES_FEISTEL(block, subkeys, mode)
    left ← Extract the first 32 bits of block
    right ← Extract the last 32 bits of block
    for i ← 0 to DES_ROUNDS − 1 do
        temp ← right
        if mode == DES_ENCRYPT then
            right ← des_ffunc(right, subkeys[i])
        else if mode == DES_DECRYPT then
            right ← des_ffunc(right, subkeys[DES_ROUNDS − i − 1])
        end if
        right ← left ⊕ right
        left ← temp
    end for
    return concat(right, left)
end procedure
procedure DES_FFUNC(subblock, subkey)
    expanded_subblock ← des_fexpand(subblock)
    expanded_subblock ← expanded_subblock ⊕ subkey
    expanded_subblock ← des_sbox(subblock)
    subblock ← des_fpermutation(subblock)
    return subblock
end procedure
```

4.1.3 Usage example

Before using the encryption function, we need to use PKCS#7 padding (to ensure block size is an exact multiple of 64 bits) and then convert the text input to blocks of hex. Similarly, after decryption, we have to unpadding and convert hex blocks back to the text if necessary.

```
uint64_t key = 0x133457799BBCDFF1;
char *message = "Hello world!\0";
size_t len = strlen(message);

// Pad input if necessary
size_t padded_len;
unsigned char *padded_message = pkcs7_padding((unsigned char *)message, len, &padded_len, DES_BLOCK_SIZE);

// Convert text message to blocks in hex
...
// Encrypt message/file
des(input_blocks, encrypted, key, len_blocks, DES_ENCRYPT);
des_file(input_file, encrypted_file, key, DES_ENCRYPT);
// Decrypt message/file
des(encrypted, decrypted, key, len_blocks, DES_DECRYPT);
des_file(encrypted_file, decrypted_file, key, DES_DECRYPT);

//Remove padding
unsigned char *unpadded_message =
↪  pkcs7_unpadding(decrypted_message,padded_len,&decrypted_len,DES_BLOCK_SIZE);
```

4.2 AES Encryption

4.2.1 Background

AES (Advanced Encryption Standard) is a symmetric-key encryption algorithm adopted as a federal standard by NIST in 2001. Designed to replace DES, AES is faster, more secure, and supports larger key sizes (128, 192, or 256 bits).

AES is widely used in applications like SSL/TLS, disk encryption, and secure communications due to its robustness against modern cryptographic attacks.

Unlike DES, AES supports multiple modes of operation, which define how the algorithm processes data larger than a single block. Modes are available in this library: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). There are also functions for file encryption.

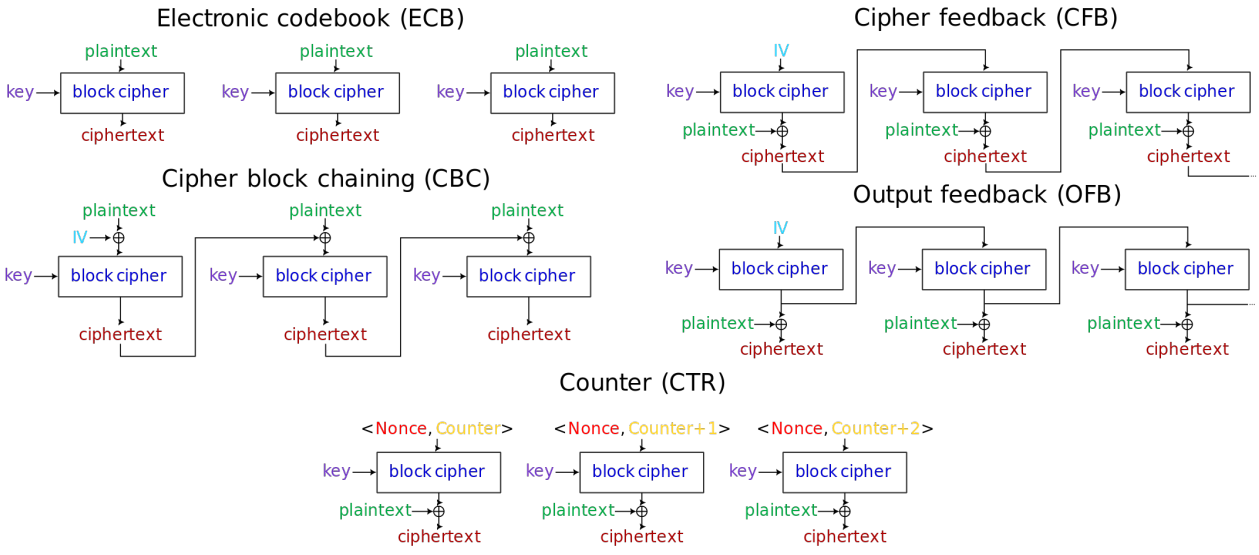


Figure 2: Implemented AES encryption modes [RE24]

4.2.2 Overall structure

The library consists of 3 main files:

- The header file (aes.h): defines the constants, data types, and functions of the AES encryption algorithm.
- The source file (aes.c): provides the implementations for all of the defined functions.
- The test file (aes.test.c): gives an example of how to use the AES encryption library.

Similar to DES, AES also has a key schedule step to expand the original into several round keys (subkeys or words). These round keys are used in each round of the AES encryption and decryption process.

Key schedule steps:

- The first key_size bytes of the expanded key (16, 24, or 32 bytes) are copied directly from the initial key.
- The remaining round keys are generated iteratively using the following steps:
  - RotWord: A 4-byte word is rotated left by one byte.
  - SubWord: Each word byte is substituted using the AES S-box.
  - Rcon: A round constant is XORed with the word to introduce asymmetry.
  - XOR with Previous Word: The transformed word is XORed with the word N bytes earlier in the expanded key.

Algorithm 9 AES KeySchedule

Input: Key key, Key Size key_size, Number of Round $rounds$
Output: $words$: Expanded keys

Initialize the key schedule $words \leftarrow key$
for $i \leftarrow key\_size$ **to** $4 \times 4 \times (rounds + 1)$ **do**
  for $j \leftarrow 0$ **to** 4 **do**
    Copy the last 4 bytes (1 word) from the schedule into $temp[j] \leftarrow words[i - 4 + j]$
  end for
  if $i \mod key\_size == 0$ **then**
    $temp \leftarrow aes\_rot\_word(temp)$
    $temp \leftarrow aes\_sub\_word(temp)$
    $rcon \leftarrow aes\_rcon(rcon, i/key\_size)$
    $temp \leftarrow aes\_xor\_words(temp, rcon)$
  else if $key\_size > 24$ && $i \mod key\_size == 16$ **then**
    For AES-256, substitute the word at every 32 bytes $temp \leftarrow aes\_sub\_word(temp)$
  end if
  for $j \leftarrow 0$ **to** 4 **do**
    $words[i + j] \leftarrow words[i - key\_size + j] \bigoplus temp[j]$
  end for
  $i \leftarrow i + 4$
end for
return $words$

AES operates on a 4×4 column-major order array of 16 bytes (State). Each round consists of several processing steps, including one that depends on the encryption key itself. Like DES, encryption and decryption only differ in the order of the round keys and the inverses of some transformation functions (SubBytes - InvSubBytes, ShiftRows - InvShiftRows, MixColumns - InvMixColumns).[AES23]

Algorithm 10 AES Block Encryption

Input: Expanded keys *words*
Output: *block*: Block to be encrypted

state ← Put *block* into 4×4 bytes 2D-array
state ← aes_add_round_key(state, words)
for $i \leftarrow 0$ **to** $AES\_ROUNDS - 1$ **do**
    $state \leftarrow aes\_sub\_bytes(state)$
    $state \leftarrow aes\_shift\_rows(state)$
    $state \leftarrow aes\_mix\_columns(state)$
    $state \leftarrow aes\_add\_round\_key(state, words + i * 16)$
end for
$state \leftarrow aes\_sub\_bytes(state)$
$state \leftarrow aes\_shift\_rows(state)$
$state \leftarrow aes\_add\_round\_key(state, words + AES\_ROUNDS * 16)$
block ← Get *state* out
return *block*

Each round processes the data as follows:

- SubBytes: Non-linear byte substitution using the AES_SBOX.
- ShiftRows: Cyclically shifts rows of the state matrix.
- MixColumns: Mixes columns using $GF(2^8)$ multiplication.
- AddRoundKey: XORs the state with the current round key.

4.2.3 Usage example

Similar to DES, we need to use PKCS#7 padding (to ensure block size is an exact multiple of 64 bits) and then convert the text input to blocks of hex. Similarly, after decryption, we have to unpadding and convert hex blocks back to the text if necessary.

```
char *key = "133457799BBCDFF1133457799BBCDFF1";
char message[] = "Hello world! My name is John Doe.\0";
size_t len = strlen((const char *)message);

// Pad input if necessary
unsigned char *padded_message = pkcs7_padding((unsigned char *)message, len, &padded_len, AES_BLOCK_SIZE);

// Convert text message to blocks in hex
...
// Encrypt message/file
aes_encrypt_ecb(input_blocks, encrypted_ecb, key_hex, padded_len, AES_KEY_SIZE_128);
aes_file_encrypt(input_file, encrypted_file, key_hex, iv_file, AES_KEY_SIZE_128, AES_MODE_CBC);
// Decrypt message/file
aes_decrypt_ecb(encrypted_ecb, decrypted_ecb, key_hex, padded_len, AES_KEY_SIZE_128);
aes_file_decrypt(encrypted_file, decrypted_file, key_hex, iv_file, AES_KEY_SIZE_128, AES_MODE_CBC);

//Remove padding
unsigned char *unpadded_message = pkcs7_unpadding(decrypted_ecb, padded_len, &decrypted_len, AES_BLOCK_SIZE);
```

4.3 CBC-MAC

4.3.1 Background

A Cipher block chaining message authentication code (CBC-MAC) is a technique for constructing a message authentication code (MAC) from a block cipher. It combines the structure of Cipher Block Chaining (CBC) mode encryption with a focus on generating a unique fixed-size message authentication code (MAC) rather than encrypting data. The primary application of CBC-MAC is in securing messages against tampering or forgery.

CBC-MAC is widely used for ensuring data integrity and authenticity in secure communication protocols like IPsec, TLS, and WPA2, as well as in payment systems like EMV cards, firmware validation, IoT device communication, and database transaction integrity.

4.3.2 Overal structure

The CBC-MAC algorithm uses a symmetric block cipher (AES in this case) as its core building block. The MAC is generated by processing the input message block-by-block using the CBC mode, where each block depends on the previous block's ciphertext.

There are three files in our CBC-MAC library:

- The header file (cbc_mac.h): defines the constants, data types, and functions of the CBC-MAC algorithm.
- The source file (cbc_mac.c): provides the implementations for all of the defined functions.
- The test file (cbc_mac.test.c): gives an example of how to use the CBC-MAC library.

Using AES encryption in CBC mode, the CBC-MAC algorithm can compute the MAC by extracting the last block after encryption. As mentioned before, AES needs proper padding with PKCS#7 before execution.

Algorithm 11 CBC-MAC

Input: Key k, Key Size $keysize$, Data $data$, Data Length $data\_len$, Security Level sec_level
Output: MAC mac

1: Determine key_len based on $sec\_level : key\_len(128, 192, 256) \leftarrow sec\_level(L1, L2, L3)$
2: Pad $data$ using PKCS#7 to align with AES block size, obtaining $padded\_message$ and $padded\_len$
3: Initialize $key\_block \leftarrow k$ (trim or pad k to AES block size)
4: Set Initialization Vector $iv \leftarrow 0$
5: Allocate memory for $input\_blocks$ and $encrypted\_ecb$
6: **for** $i = 1$ to $padded\_len$ **do**
7: Copy $padded\_message[i]$ to $input\_blocks[i]$
8: **end for**
9: Encrypt $input\_blocks$ using CBC mode $encrypted\_ecb \leftarrow$ aes_encrypt_cbc$(input_blocks, iv, key_block, key_len)$
10: Extract last block $mac \leftarrow encrypted\_ecb[padded\_len - AES\_BLOCK\_SIZE]$
11: **return** mac

To verify the integrity and authenticity of the input data, the algorithm computes a new MAC and compares it with the provided MAC.

Algorithm 12 CBC-MAC Verification

Input: Key k, Key Size $keysize$, Data $data$, Data Length $data\_len$, Security Level sec_level, Provided MAC mac
Output: Verification Result

1: Compute mac_check using **Algorithm 6 (CBC-MAC Calculation)** on $data$
2: Compare mac_check with mac
3: **if** $mac = mac\_check$ **then return** 0 (Valid)
4: **else** **return** Non-zero (Invalid)
5: **end if**

4.4 HMAC

4.4.1 Background

A Hash-based Message Authentication Code (HMAC) is a method used for ensuring both the integrity and authenticity of data. HMAC combines a cryptographic hash function (MD5, SHA1, SHA2, or SHA3 in this case) with a shared secret key to generate a fixed-size message authentication code (MAC). This approach protects against data tampering and impersonation.

HMAC is widely applied in secure communication protocols such as TLS, IPsec, and SSH. It is also used in web API authentication (e.g., AWS signatures), database integrity checks, and digital signature systems to enhance security against attacks like message forgery or replay.

4.4.2 Overall structure

The HMAC library has three key files:

- Header file (hmac.h): Declares constants, data structures, and HMAC-related functions.
- Source file (hmac.c): Implements the HMAC operations (initialization, update, finalization, and verification).
- Test file (hmac.test.c): Demonstrates how to use the HMAC library with practical examples.

HMAC is defined as:

$$HMAC(K, m) = H((K' \oplus opad)||H((K' \oplus ipad)||m))$$

with H: a cryptographic hash function, K': key padded to block size, $ipad$: inner padding ($0x36$ repeated), $opad$: outer padding ($0x5c$ repeated).

Algorithm 13 HMAC

Input: Key key, Key Size $keylen$, Data $data$, Data Length $data\_len$, Security Level sec_level, Hash Function $hash$

Output: mac: MAC

1: **Initialize HMAC context** with key, key_len, sec_level, and $hash$
2: **Key Padding**:
3: **if** $key\_len > BLOCK\_SIZE$ **then**
4: Hash the key k and truncate to the block size
5: **else**
6: Pad k with zeros to make it $BLOCK\_SIZE$ bytes
7: **end if**
8: Compute $i\_key\_pad \leftarrow k \oplus 0x36$ (inner padding)
9: Compute $o\_key\_pad \leftarrow k \oplus 0x5c$ (outer padding)
10: **Inner Hash:**
11: Concatenate i_key_pad with $data$: $inner \leftarrow i\_key\_pad \| data$
12: Compute $inner\_hash \leftarrow hash(inner)$
13: **Outer Hash:**
14: Concatenate o_key_pad with $inner\_hash$: $outer \leftarrow o\_key\_pad \| inner\_hash$
15: Compute $mac \leftarrow hash(outer)$
16: **return** mac

To verify the provided MAC, we compute a new MAC with the given data and then compare them.

Algorithm 14 HMAC Verification

Input: Key k, Key Size key_len, Data $data$, Data Length $data_len$, Hash Function $hash$, Security Level sec_level, Provided MAC mac

Output: Verification Result

1: Compute mac_check using the **HMAC Algorithm** on $data$
2: Compare mac_check with mac
3: **if** $mac = mac\_check$ **then return** 0 (Valid)
4: **else return** Non-zero (Invalid)
5: **end if**

5 Public Key Cryptography

5.1 RSA Cryptosystem

5.1.1 Background

RSA is one of the first and most widely used public-key cryptosystems, primarily used for secure data transmission. It was developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at the Massachusetts Institute of Technology (MIT). The algorithm is based on the mathematical difficulty of factoring large prime numbers, which forms the foundation of its security.

- Key Aspects of RSA's Background
  - Public-Key Cryptography:
    - Before RSA, encryption methods were primarily symmetric, meaning the same key was used for both encryption and decryption. RSA introduced asymmetric cryptography, where a public key is used for encryption, and a private key is used for decryption.
  - Mathematical Foundation:
    - RSA relies on the factoring problem, which is the difficulty of factoring a large number that is the product of two large prime numbers.
    - The security of RSA is based on the assumption that, while multiplication of two primes is easy, factoring the resulting large number is computationally infeasible with current technology.
  - Patent and Public Release:
    - The RSA algorithm was patented in the United States (US Patent 4,405,829) in 1983.
    - The patent expired in 2000, making RSA freely available for use worldwide.
    - The RSA cryptosystem is one of the earliest and most widely used public-key cryptosystems. It is named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who first publicly described the algorithm in 1977 [RSA78].
- We implemented three variants for both RSA public key encryption and digital signature, all of them are specified in the RSA standard RFC8017 [MKJR16].
  - Textbook RSA: The most simple version of RSA algorithms whose security is usually known to be reduced to the hardness assumption of the integer factorization problem. This variant is taught in every basic course on public key cryptography but is nowhere near to the current standard of security.
  - PCKS#1 v1.5: It is the combination of the RSA primitives with the EME-PKCS1-v1_5 encoding method, proposed in [Kal98]. While it had been widely adopted, e.g. implemented in `OpenSSL`, it is not recommended in RFC8017.
  - OAEP & PSS: These two variants are the most secure variants of RSA public encryption and digital signature, each equipped with its own encoding scheme and utilizes MGF1, an MGF - Mask Generation Function, in its algorithm.

  In the following sections, we will describe only the basic primitives of RSA, the details of variants' algorithms can be found in the appendix F.

5.1.2 Overall structure

- The header file (rsa.h): defines the constants, data types, and functions of the RSA encryption and signature algorithm.
- The encryption source file (enc.c): provides the implementations for all of the encryption-defined functions.
- The encryption test file (enc.test.c): gives an example of how to use the RSA encryption library.
- The signature source file (sig.c): provides the implementations for all of the signature-defined functions.
- The signature test file (sig.test.c): gives an example of how to use the RSA signature library.

Here's a simplified overview of how RSA works:

- **Key Generation**: RSA begins with the generation of two large random prime numbers, which are then multiplied together to produce a number called the modulus. The public and private keys are created using this modulus. The public key also includes an exponent, which is typically a small number like 65537. The private key includes a different, carefully calculated exponent.
- **Encryption**: When someone wants to send a secure message to the owner of the private key, they encrypt the message using the recipient's public key. The encryption involves raising the message to the power of the public exponent and then taking the remainder when divided by the modulus.
- **Decryption**: The recipient can decrypt the message using the private key. This process involves raising the encrypted message to the power of the private exponent and again taking the remainder when divided by the modulus.

```c
// Security parameters for different security levels
#define L0_N_BITS 2048
#define L0_E "65537" // Common RSA public exponent

#define L1_N_BITS 3072
#define L1_E "65537"

#define L2_N_BITS 7680
#define L2_E "65537"

#define L3_N_BITS 15360
#define L3_E "65537"

typedef enum
{
    RSA_STANDARD,
    RSA_CRT
} rsa_algo_t;

typedef struct public_params
{
    int n_bits;
    mpz_t e;
} public_params_t;

typedef struct pub_key
{
    mpz_t n; // modulus
    mpz_t e; // public exponent
} pub_key_t;

typedef struct priv_key
{
    mpz_t n;     // modulus
    mpz_t p;     // first prime factor
    mpz_t q;     // second prime factor
    mpz_t d;     // private exponent
    mpz_t dp;    // d mod (p-1)
    mpz_t dq;    // d mod (q-1)
    mpz_t q_inv; // q^(-1) mod p
} priv_key_t;

// Core RSA functions
void setup(public_params_t *pp, sec_level_t level);

void keygen(priv_key_t *sk, pub_key_t *pk, public_params_t pp);

void crypto_encrypt(mpz_t c, const mpz_t m, const pub_key_t *pk);
void crypto_decrypt(mpz_t m, const mpz_t c, const priv_key_t *sk, rsa_algo_t algorithm);
void crypto_sign(mpz_t s, const mpz_t m, const priv_key_t *sk, rsa_algo_t algorithm);
int crypto_verify(const mpz_t m, const mpz_t s, const pub_key_t *pk);

void crypto_encrypt_pkcs1(mpz_t c, const mpz_t m, const pub_key_t *pk);
void crypto_decrypt_pkcs1(mpz_t m, const mpz_t c, const priv_key_t *sk, rsa_algo_t algorithm);
```

```
55 void crypto_sign_pkcs1(mpz_t s, const mpz_t m, const priv_key_t *sk, rsa_algo_t algorithm, sec_level_t
   ↪ sec_level);
56 int crypto_verify_pkcs1(const mpz_t m, const mpz_t s, const pub_key_t *pk, sec_level_t sec_level);
57
58 void crypto_encrypt_oaep(mpz_t c, const mpz_t m, const pub_key_t *pk, sec_level_t sec_level);
59 void crypto_decrypt_oaep(mpz_t m, const mpz_t c, const priv_key_t *sk, rsa_algo_t algorithm, sec_level_t
   ↪ sec_level);
60 void crypto_sign_pss(mpz_t s, const mpz_t m, const priv_key_t *sk, rsa_algo_t algorithm, sec_level_t
   ↪ sec_level);
61 int crypto_verify_pss(const mpz_t m, const mpz_t s, const pub_key_t *pk, sec_level_t sec_level);
```

Explanation of the rsa.h file code

- The **typedef struct** sequence defines structures used for RSA key management and parameters. These include:
  - **public_params_t**: Holds the security level parameters, consisting of an integer for key size (**n_bits**) and the public exponent (**e**).
  - **pub_key_t**: Defines the public key structure containing the modulus (**n**) and the public exponent (**e**).
  - **priv_key_t**: Defines the private key structure with:
    - **mpz_t n**: The RSA modulus.
    - **mpz_t p, q**: The prime factors of **n**.
    - **mpz_t d**: The private exponent.
    - **mpz_t dp, dq**: The values for optimized decryption using CRT (Chinese Remainder Theorem).
    - **mpz_t q_inv**: The modular inverse of **q** modulo **p**.
- **Function Prototypes**: The header file declares various RSA-related functions, including:
  - **void setup(public_params_t *pp, sec_level_t level)**: Initializes security parameters based on the desired security level.
  - **void keygen(priv_key_t *sk, pub_key_t *pk, public_params_t pp)**: Generates an RSA key pair.
  - **void crypto_encrypt(mpz_t c, const mpz_t m, const pub_key_t *pk)**: Encrypts a message **m** using the public key.
  - **void crypto_decrypt(mpz_t m, const mpz_t c, const priv_key_t *sk, rsa_algo_t algorithm)**: Decrypts a ciphertext **c** using the private key.
  - **void crypto_sign(mpz_t s, const mpz_t m, const priv_key_t *sk, rsa_algo_t algorithm)**: Signs a message **m** with the private key.
  - **int crypto_verify(const mpz_t m, const mpz_t s, const pub_key_t *pk)**: Verifies a digital signature.

RSA Key Generation. This program is a complete implementation for generating RSA keys, ensuring that the generated primes are large and likely prime, and that the keys meet basic RSA requirements for security.

Algorithm 15 RSA Key Generation

Input: Public parameters pp with e and n_bits
Output: Private key sk and public key pk

Initialize random state $state$
Initialize all MPZ variables in sk and pk
Set $pk.e \leftarrow pp.e$
Initialize temporary variables p_1, q_1, ϕ_n, and gcd
repeat
    Generate a prime p of size $pp.n\_bits/2$ using $prime\_gen$
    repeat
        Generate a second prime q of size $pp.n\_bits/2$ using $prime\_gen$
    until $p \neq q$
    Compute $sk.n \leftarrow p \times q$
    Set $pk.n \leftarrow sk.n$
    Compute $\phi_n \leftarrow (p-1) \times (q-1)$
    Compute $\gcd(pk.e, \phi_n)$
until $\gcd(pk.e, \phi_n) = 1$
Compute private exponent $sk.d \leftarrow pk.e^{-1} \mod \phi_n$
Compute CRT components:
$sk.dp \leftarrow sk.d \mod (p-1)$
$sk.dq \leftarrow sk.d \mod (q-1)$
$sk.q\_inv \leftarrow q^{-1} \mod p$
Clear temporary variables and random state
return sk and pk

RSA Encryption. The program expects two command-line arguments: a filename for the public key and a numeric message to encrypt. The output is the encrypted message displayed on the console, indicating the message has been encrypted with the provided RSA public key.

Algorithm 16 RSA Encryption Procedure (RSAEP)

Input: Message m, public key pk with e and n
Output: Ciphertext c

Compute $c \leftarrow m^e \mod n$
return c

RSA Decryption. The program requires two command-line arguments: the path to the private key file and the numeric cipher text. The output is the decrypted message, indicating that the cipher has been decrypted with the RSA private key provided.

Algorithm 17 RSA Decryption Procedure (RSADP)

Input: Ciphertext c, private key sk, RSA algorithm type $algo$
Output: Decrypted message m

if $algo =$ RSA_STANDARD **then**
  Compute $m \leftarrow c^d \mod n$
else if $algo =$ RSA_CRT **then**
  Initialize temporary variables: m_1, m_2, h
  Compute $m_1 \leftarrow c^{dp} \mod p$
  Compute $m_2 \leftarrow c^{dq} \mod q$
  Compute $h \leftarrow q\_inv \cdot (m_1 - m_2) \mod p$
  if $h < 0$ **then**
    $h \leftarrow h + p$
  end if
  Compute $h \leftarrow h \cdot q$
  Compute $m \leftarrow m_2 + h$
  Clear temporary variables
else
  Error: Invalid RSA algorithm type
  Exit with failure
end if
return m

RSA Signature. The program must be executed with two command-line arguments: the path to the private key file and the message to sign (expressed as a number). The output is the digital signature, which can be used for verification to confirm the message's authenticity.

Algorithm 18 RSA Signature Primitive (RSASP1)

Input: Message m, private key sk, RSA algorithm type $algorithm$
Output: Signature s

if $algorithm =$ RSA_STANDARD **then**
  Compute $s \leftarrow m^d \mod n$
else if $algorithm =$ RSA_CRT **then**
  Initialize temporary variables: s_1, s_2, h
  Compute $s_1 \leftarrow m^{dp} \mod p$
  Compute $s_2 \leftarrow m^{dq} \mod q$
  Compute $h \leftarrow q\_inv \cdot (s_1 - s_2) \mod p$
  if $h < 0$ **then**
    $h \leftarrow h + p$
  end if
  Compute $h \leftarrow h \cdot q$
  Compute $s \leftarrow s_2 + h$
  Clear temporary variables
else
  Error: Invalid RSA algorithm type
  Exit with failure
end if
return s

RSA Verification. The program can be used for verifying RSA digital signatures. The program checks the validity of a given signature for a specific message using the RSA public key.

Algorithm 19 RSA Verification Primitive (RSAVP1)

Input: Signature s, public key pk
Output: Message m

1: Compute $m \leftarrow s^e \mod n$
2: **return** m

Algorithm 20 RSA Signature Verification

Input: Message m, signature s, public key pk
Output: **True** if m is valid, **False** otherwise

1: Initialize m_{check}
2: RSAVP1(m_{check}, s, pk)
3: **if** $m == m_{\text{check}}$ **then**
4: **return True**
5: **else**
6: **return False**
7: **end if**

5.1.3 Usage example

Usage example of RSA cryptosystems can be found in the two executable test file `enc.test.c` and `sig.test.c`.

5.2 ElGamal Cryptosystem

The ElGamal asymmetric cryptosystem includes a public key encryption scheme and a digital signature scheme. It was proposed by Taher Elgamal in 1985 [Elg85], with inspiration from the Diffie-Hellman Key Exchange protocol. Nowadays, while the components in the ElGamal family are not standardized, they are still implemented in many open-source cryptographic libraries, described in almost every textbook, and provide specific functionalities for special use cases, e.g. homomorphic property in digital voting systems.

5.2.1 ElGamal Encryption

The ElGamal public key encryption scheme is a quadruple of functions `(Setup, KeyGen, Encrypt, Decrypt)`. Our implementation follows the description of the ElGamal public key encryption scheme in Chapter 8 and the ElGamal digital signature scheme in Chapter 11 of the book *Handbook of Applied Cryptography* [MVO96].

Setup. The encryption scheme is defined over any cyclic group G, e.g. multiplicative group of prime-power modulo n. In our implementation, instead of generating a random valid group, we utilized modular exponential Diffie-Hellman groups specified in RFC4525 [KK03]. The motivation is that the security of the ElGamal encryption scheme depends on the security of the Diffie-Hellman protocol, hence, using standardized groups for the key exchange task will ensure the security requirements. Therefore, the `Setup` algorithm simply returns (p, g), the constant value of modulo p and generator g for the expected level of security. See J for the constant values.

KeyGen. With the public parameters setup, Alice can generate an asymmetric key pair with the following algorithm. The random functionality is performed with `rng` module.

Algorithm 21 ElGamal Key Generation

Input: Public parameters: $pp = (p, g)$
Output: Private key x, Public key y

1: **Generate** random integer x in range $[0, p-1]$
2: **Compute** $y \leftarrow g^x \mod p$
3: **return** (x, y)

Encrypt. Then, with the public parameters and Alice's public key, Bob can encrypt a message m. As we can see, the limitation here is that the message after being conversed to an integer can not be greater than modulo p. Also, this encryption algorithm is non-deterministic, which is important for any modern encryption scheme to be considered secure.

Algorithm 22 ElGamal Encryption

Input: Message m, Public key y, Public parameters $pp = (p, g)$
Output: Ciphertext (c_1, c_2)
1: **Generate** random integer k in range $[0, p-1]$
2: **Compute** $c_1 \leftarrow g^k \mod p$
3: **Compute** $c_2 \leftarrow y^k \cdot m \mod p$
4: **return** (c_1, c_2)

Decrypt. Finally, Alice can decrypt the message with public key x and Bob's ciphertext.

Algorithm 23 ElGamal Decryption

Input: Ciphertext (c_1, c_2), Private key x, Public parameters $pp = (p, g)$
Output: Message m
1: **Compute** $s \leftarrow c_1^x \mod p$
2: **Compute** $s^{-1} \leftarrow s^{-1} \mod p$
3: **Compute** $m \leftarrow c_2 \cdot s^{-1} \mod p$
4: **return** m

5.2.2 ElGamal Signature

The ElGamal signature scheme is a randomized signature mechanism described by a quadruple of functions `(Setup, KeyGen, Sign, Verify)`. It generates digital signatures on binary messages of arbitrary length, thus requires a hash function $h : \{0,1\}^* \to \mathbb{Z}_p$ where p is a large prime number. For our implementation, we fixed the usage of hash functions from `SHA-2` family. Public parameters, public key, and private key are identical to the ElGamal encryption scheme, therefore, the `Setup` and `KeyGen` functions can be reused from the encryption implementation.

Sign. The signing message of arbitrary length is hashed into a fixed-length digest for the signing algorithm, which does not only remove the limitation of message length compared to the encryption scheme but also ensures the integrity requirement for the digital signature scheme.

Algorithm 24 ElGamal Signature Generation

Input: Message m, Private key x, Public parameters $pp = (p, g)$, Security level sec_level
Output: Signature (r, s)
1: **Set** hash length $hash\_len$ based on sec_level
2: **Compute** $p_1 \leftarrow p - 1$
3: **Generate** random integer k in range $[0, p_1 - 1]$ such that k is invertible modulo p_1
4: **Compute** $r \leftarrow g^k \mod p$
5: **Compute** hash $h \leftarrow H(m)$ using $hash\_len$-bit hash function
6: **Compute** $s \leftarrow k^{-1} \cdot (h - r \cdot x) \mod p_1$
7: **return** (r, s)

Verify. If the signature was generated by the signer, then: $s \equiv k^{-1}(h - r \cdot x) \mod (p-1)$. Multiplying both sides by k gives: $ks \equiv h - r \cdot x \mod (p-1)$, and rearranging yields:$h \equiv r \cdot x + ks \mod (p-1)$. This implies: $g^h \equiv g^{r \cdot x + ks} \equiv (g^x)^r \cdot r^s \mod p$. Thus, $v_1 = v_2$, as required.

Algorithm 25 ElGamal Signature Verification

Input: Message m, Signature (r, s), Public key y, Public parameters $pp = (p, g)$, Security level sec_level
Output: **Valid** if signature is correct, otherwise **Invalid**

Set hash length $hash\_len$ based on sec_level
Compute hash $h \leftarrow H(m)$ using $hash\_len$-bit hash function
Compute $v_1 \leftarrow y^r \cdot r^s \mod p$
Compute $v_2 \leftarrow g^h \mod p$
if $v_1 = v_2$ **then**
    return Valid
else
    return Invalid
end if

An adversary might attempt to forge a signature on m by selecting a random integer k and computing: $r = g^k \mod p$. The adversary must then determine: $s = k^{-1}(h - r \cdot x) \mod (p - 1)$. If the discrete logarithm problem is computationally infeasible, the adversary can do no better than to choose s at random; the success probability is only $\frac{1}{p}$, which is negligible for large p.

```c
typedef struct public_params
{
    int p_bits;
    mpz_t p;
    mpz_t g;
} public_params_t;

typedef struct pub_key
{
    mpz_t y;
} pub_key_t;

typedef struct priv_key
{
    mpz_t x;
} priv_key_t;

void setup(public_params_t *pp, sec_level_t level);
void keygen(priv_key_t *sk, pub_key_t *pk, const public_params_t *pp);
void crypto_encrypt(mpz_t c1, mpz_t c2, const mpz_t m, const pub_key_t *pk, const public_params_t *pp);
void crypto_decrypt(mpz_t m, const mpz_t c1, const mpz_t c2, const priv_key_t *sk, const public_params_t
↪  *pp);
void crypto_sign(mpz_t r, mpz_t s, const unsigned char *m, size_t m_len, const priv_key_t *sk, const
↪  public_params_t *pp, sec_level_t sec_level);
int crypto_verify(const mpz_t r, const mpz_t s, const unsigned char *m, size_t m_len, const pub_key_t *pk,
↪  const public_params_t *pp, sec_level_t sec_level);
```

The ElGamal cryptosystem defined in `elgamal.h` is implemented in two separated source files: `enc.c` and `sig.c`, each has its own corresponding executable test files.

5.3 DSA - Digital Signature Algorithm

The DSA is a digital signature scheme proposed by NIST in 1991, then later standardized in FIPS 186 [NIS13a] by the same organization. While it has been adopted widely since its standardization, after five revisions of FIPS 186, the DSA signature scheme has been disapproved by NIST. Similar to other digital signature schemes, DSA is defined as a quadruple of functions `(Setup, KeyGen, Sign, Verify)`.

Setup. In the FIPS 186 standard, public parameters of the signature scheme are called domain parameters, which shall be generated with a specific algorithm for the generation of provable primes using an approved hash function. However, this setup task would take quite some time and we decided to utilize fixed domain parameters from examples of NIST implementation for our implementation [NIS13b]. We argued that this decision would be more simple and fit the purpose of demonstration rather than a production-level quality of this project.

KeyGen. From the selected/generated parameters, Alice can generate an asymmetric key pair for the signature scheme.

Algorithm 26 DSA Key Generation

Input: Public parameters $pp = (p, q, g)$
Output: Private key x, Public key y

1: **Generate** random integer x in range $[0, q-1]$
2: **Compute** $y \leftarrow g^x \mod p$
3: **return** (x, y)

Sign. The signing algorithm requires a hash function $h : \{0,1\}^* \to \mathbb{Z}_q$ to sign messages of arbitrary length. In our implementation, we fixed the usage of hash functions from `SHA-2` or `SHA-3` family.

Algorithm 27 DSA Signature Generation

Input: Message m, Private key x, Public parameters $pp = (p, q, g, H)$
Output: Signature (r, s)

1: **Generate** random integer k in range $[0, q-1]$
2: **Compute** hash $h_m \leftarrow H(m)$
3: **Compute** $r \leftarrow g^k \mod p$
4: **Compute** $r \leftarrow r \mod q$
5: **Compute** $s \leftarrow k^{-1}(h_m + x \cdot r) \mod q$
6: **return** (r, s)

Verify. Given a signature (r, s), the verifier computes: $w = s^{-1} \mod q$, $u_1 = h_m \cdot w \mod q$, $u_2 = r \cdot w \mod q$. Then, the verifier computes: $v = g^{u_1} \cdot y^{u_2} \mod p$. From the signing equation, we have: $s \equiv k^{-1}(h_m + x \cdot r) \mod q$. Multiplying both sides by $w = s^{-1} \mod q$ gives: $1 \equiv k \cdot w \cdot (h_m + x \cdot r) \mod q$. Rearranging: $k \cdot w \cdot h_m + k \cdot w \cdot x \cdot r \equiv 1 \mod q$. Since $u_1 = h_m \cdot w \mod q$ and $u_2 = r \cdot w \mod q$, we rewrite: $g^{h_m \cdot w} \cdot y^{r \cdot w} \equiv g^{h_m \cdot w} \cdot (g^x)^{r \cdot w} \mod p$. Using exponent properties: $g^{h_m \cdot w} \cdot g^{x \cdot r \cdot w} \equiv g^{(h_m + x \cdot r) \cdot w} \equiv g^k \mod p$. Since $r = g^k \mod p$, we obtain: $v = r$.

Algorithm 28 DSA Signature Verification

Input: Message m, Signature (r, s), Public key y, Public parameters $pp = (p, q, g, H)$
Output: **Valid** if signature is correct, otherwise **Invalid**

1: **if** $r \notin [1, q-1]$ or $s \notin [1, q-1]$ **then**
2: **return Invalid**
3: **end if**
4: **Compute** hash $h_m \leftarrow H(m)$
5: **Compute** $w \leftarrow s^{-1} \mod q$
6: **Compute** $u_1 \leftarrow h_m \cdot w \mod q$
7: **Compute** $u_2 \leftarrow r \cdot w \mod q$
8: **Compute** $v \leftarrow g^{u_1} \cdot y^{u_2} \mod p$
9: **if** $v = r$ **then**
10: **return Valid**
11: **else**
12: **return Invalid**
13: **end if**

Our implementation was tested with the successful verification of signatures on random messages.

```c
typedef struct public_params
{
    int p_bits;
    int q_bits;
    mpz_t p;
    mpz_t q;
    mpz_t g;
```

```
    size_t md_len;
    void *(*hash)(const void *, size_t, void *, size_t);
} public_params_t;

typedef struct pub_key
{
    mpz_t y;
} pub_key_t;

typedef struct priv_key
{
    mpz_t x;
} priv_key_t;

void setup(public_params_t *pp, sec_level_t level, hash_func_t hash);
void keygen(priv_key_t *sk, pub_key_t *pk, public_params_t pp);
void crypto_sign(mpz_t r, mpz_t s, const unsigned char *m, int len, priv_key_t sk, public_params_t pp);
char crypto_verify(const mpz_t r, const mpz_t s, const unsigned char *m, int len, pub_key_t pk,
↪  public_params_t pp);
```

5.4 Diffie-Hellman Key Exchange

Diffie-Hellman key exchange protocol, published by Whitfield Diffie and Martin Hellman in 1976 [DH76], is one of the most important works that open the world of public key cryptography. Essentially, it helps establish a shared secret between two parties that can be used for secret communication for exchanging data over a public network, which nowadays has become the standard for key exchange between entities over the internet. We implemented the most simple version of DH Key Exchange based on the discrete logarithm problem on multiplicative groups of prime modulo. The protocol can be described with four functionalities `(Setup, GenSecret, GenPublic, GenSharedSecret)`.

Setup. We once again utilized the modular exponential groups proposed by RFC3526 [KK03] for better security.

GenSecret. Alice and Bob start by generating their secret exponents.

Algorithm 29 DH Secret Generation

Input: Public parameters $pp = (p, g)$
Output: Secret key s
1: **Generate** random integer s in range $[0, p-1]$
2: **return** s

GenPublic. Then, both of them continue to generate the corresponding public value by to send to the other.

Algorithm 30 DH Public Computation

Input: Secret key s, Public parameters $pp = (p, g)$
Output: Public key P
1: **Compute** $P \leftarrow g^s \mod p$
2: **return** P

GenSharedSecret. The final shared secret value can be retrieved from both sides by using their own secret value with the other's public value.

Algorithm 31 DH Shared Secret Computation

Input: Private key s, Other party's public key P, Public parameters $pp = (p, g)$
Output: Shared secret K
1: **Compute** $K \leftarrow P^s \mod p$
2: **return** K

While the correctness is trivial, we noted that this simple implementation of DH key exchange is vulnerable to different attacks, e.g. Man-in-the-Middle attacks.

6 Elliptic Curve Cryptography

Elliptic-curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. Proposed independently by Neal Koblitz and Victor S. Miller in 1985, the use of elliptic curves in cryptography brought to live exciting alternatives for previous public key cryptography schemes initiated from the 70s such as RSA, ElGamal, etc. The interesting part is that ECC is equipped with smaller keys for equivalent security, particularly when compared to cryptosystems based on modular exponentiation or integer factorization. In the 2000s, ECC schemes have become widely adopted with various standardization, e.g. ECDSA, EC-ElGamal, ECDH, etc. as well as implementation in cryptographic libraries. With the recent rise of quantum computing power and algorithms, the future of ECC does not look bright. Nevertheless, the role of ECC in the field of modern cryptography is still irreplaceable for the next decade at the very least.

6.1 Elliptic Curves

In this project, we followed the *Recommandations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters* [NIS23b], a document standardized by NIST to implement the three main families of elliptic curves used in cryptosystems: Weierstrass curves, Montgomery curves, and Twisted Edwards curves.

Security Level	Elliptic Curves
L0	P-224
L1	P-256, W-25519, Curve25519, Edwards25519
L2	P-384, W-448, Curve448, Edwards448, E448
L3	P-521

Table 4: Approximate Security Strength of the Implemented Curves

The implemented curves are a subset of approved curves specified in the mentioned document. We decided to represent the curve elements over prime fields rather than binary fields, and the domain parameters are set as recommended constant values rather than generated randomly for efficiency.

6.1.1 Weierstrass Curves

The implemented Weierstrass curves over prime fields are P-224, P-256, P-384, P-521, W-25519, and W-448. Each of these curves is defined in Short-Weierstrass form:

$$E : y^2 \equiv x^3 + ax + b \mod p,$$

the following domain parameters $D = (p, h, n, a, b, G$ are given:

- **The prime modulus** p.
- **The cofactor** $h > 1$.
- **The coefficient** a.
- **The coefficient** b.
- **The base point** G with x-coordinate G_x and y-coordinate G_y.

6.1.2 Montgomery Curves

The implemented Montgomery curves over prime fields are W-25519 and W-448. Each of these curves is defined in Montgomery form:

$$E : By^2 \equiv x^3 + Ax^2 + x \mod p,$$

where the following domain parameters $D = (p, h, n, A, B, G)$ are given:

- **The prime modulus** p.
- **The cofactor** $h > 1$.
- **The coefficient** A.

- **The coefficient** B.
- **The base point** G with x-coordinate G_x.

6.1.3 Edwards Curves

The implemented Twisted Edwards curves over prime fields are Curve25519 and Edwards448. Each of these curves is defined in Twisted Edwards form:

$$E : ax^2 + y^2 \equiv 1 + dx^2y^2 \mod p,$$

where the following domain parameters $D = (p, h, n, a, d, G)$ are given:

- **The prime modulus** p.
- **The cofactor** $h > 1$.
- **The coefficient** a.
- **The coefficient** d.
- **The base point** G with x-coordinate G_x and y-coordinate G_y.

We utilized `GMP`'s `mpz_t` to represent the coordinates of points on elliptic curves. These three families of elliptic curves share the same interface of functionalities:

```c
typedef struct point
{
    mpz_t x;
    mpz_t y;
    mpz_t z;
} point_t;

typedef struct point_affine
{
    mpz_t x;
    mpz_t y;
} point_affine_t;

typedef struct curve
{
    unsigned char type;
    unsigned char id;
    int cof; // or h: cofactor
    size_t efs;
    mpz_t p; // or q: field size
    mpz_t r; // or n: group order
    mpz_t a;
    mpz_t b;
    point_affine_t G;
    size_t md_len;
    void *(*hash)(const void *, size_t, void *, int);
    char *name;
} curve_t;

void init_point(point_t *p);
void free_point(point_t *p);
void copy_point(point_t *r, const point_t p);
void swap_points(point_t *p, point_t *q);

void init_affine(point_affine_t *p);
void free_affine(point_affine_t *p);
void copy_affine(point_affine_t *r, const point_affine_t p);

void init_curve(curve_t *curve, ec_t curve_type, unsigned char curve_id);
void free_curve(curve_t *curve);

void rhs(mpz_t r, const mpz_t x, const curve_t curve);
void generator(point_t *p, const curve_t curve);
```

```
void infinity(point_t *p, const curve_t curve);
char is_infinity(const point_t p, const curve_t curve);
char is_on_curve(const point_t p, const curve_t curve);
char equals(const point_t p, const point_t q, const curve_t curve);
void neg(point_t *r, const point_t p, const curve_t curve);
void add(point_t *r, const point_t p, const point_t q, const curve_t curve);
void dadd(point_t *r, const point_t p, const point_t q, const point_t w, const curve_t curve);
void dbl(point_t *r, const point_t p, const curve_t curve);
void mul(point_t *r, const point_t p, const mpz_t k, const curve_t curve);
void to_affine(point_affine_t *r, const point_t p, const curve_t curve);
void from_affine(point_t *r, const point_affine_t p, const curve_t curve);
void point_to_bytes(unsigned char *buf, const point_t p, const curve_t curve);
void point_from_bytes(point_t *r, const unsigned char *buf, const curve_t curve);
```

While most of those functionalities are highly similar, the main difference lies in the definition of point addition and point scalar multiplication. For these algorithms, we implemented the formulae specified in the *Explicit-Formulas Database* [BL]. For more explanation on the formulae, see K.

All of the implemented curves passed the test of checking the consistency of the addition and scalar multiplication operations, generator and infinity points, and conversion between affine and projective coordinates.

6.2 EC-ElGamal Cryptosystem

The EC-ElGamal cryptosystem is a variant of the ElGamal cryptosystem that utilizes the group of points on the selected elliptic curves rather than a multiplicative group of prime order modulo. Hence, the syntaxes of both the public key encryption scheme and the digital signature scheme stay unchanged, only the algorithms changed to adapt the usage of elliptic curves.

6.2.1 EC-ElGamal Encryption

Different from the ElGamal encryption scheme specified above, this elliptic curve variant requires a hash function, particularly fixed as a function from `SHA-2` or `SHA-3` family in our implementation.

Setup. The setup process will initiate the selected elliptic curve.

KeyGen. A secret scalar value is uniformly sampled and used to calculate the corresponding public point on the curve.

Algorithm 32 ECElGamal Key Generation

Input: Public parameters $pp = (p, q, G, \text{curve})$
Output: Private key x, Public key Y

1: **Generate** random integer x in range $[0, q-1]$
2: **Compute** generator point G on the curve
3: **Compute** public key $Y \leftarrow x \cdot G$
4: **return** (x, Y)

Encrypt. While some other variants of EC-ElGamal might require encoding the plaintext as a point on the base curve, our implementation is encoding-free (with the price of losing homomorphic property).

Algorithm 33 ECElGamal Encryption

Input: Message m, Public key Y, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: Ciphertext (c_1, c_2)

1: **Select** hash function H
2: **Generate** random integer k in range $[0, q-1]$
3: **Compute** ephemeral public key $C_1 \leftarrow k \cdot G$
4: **Encode** C_1 as integer c_1
5: **Compute** shared secret $S \leftarrow k \cdot Y$
6: **Compute** hash $h \leftarrow H(S)$
7: **Encrypt** message: $m' \leftarrow m \oplus h$
8: **Encode** encrypted message as integer c_2
9: **return** (c_1, c_2)

Decrypt. Given a plaintext message m, encryption is performed using a randomly selected ephemeral key k, computing: $C_1 = k \cdot G, \quad S = k \cdot Y, \quad h = H(S)$, $c_2 = m \oplus h$. Decryption recovers the shared secret using the recipient's private key: $S' = x \cdot C_1$. Since: $S' = x \cdot (k \cdot G) = k \cdot (x \cdot G) = k \cdot Y = S$, we obtain: $h' = H(S') = H(S)$, ensuring correct decryption: $m = c_2 \oplus h$. Thus, the encryption and decryption process is sound.

Algorithm 34 ECElGamal Decryption

Input: Ciphertext (c_1, c_2), Private key x, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: Decrypted message m

Select hash function H
Decode ephemeral public key C_1 from c_1
Compute shared secret $S \leftarrow x \cdot C_1$
Compute hash $h \leftarrow H(S)$
Decode encrypted message $m' \leftarrow c_2$
Decrypt message: $m \leftarrow m' \oplus h$
return m

6.2.2 EC-ElGamal Signature

Similar to the public key encryption scheme, the EC-ElGamal signature scheme also requires the selection of hash functions from `SHA-2` or `SHA-3` family. We note that the digest length, i.e. the security level of the hash function, is already implied by the security level of the selected curve.

Sign. The signing algorithm has high similarity to the ElGamal signature scheme. The signer computes the signature as follows: $s_1 \equiv R_x \mod q$ and $s_2 \equiv k^{-1}(h + s_1 \cdot x) \mod q$.

Algorithm 35 ECElGamal Signature Generation

Input: Message m, Private key x, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: Signature (s_1, s_2)

Select hash function H from {SHA-2, SHA-3}
Compute hash $h \leftarrow H(m)$
repeat
    Generate random integer k in range $[0, q-1]$
    Compute ephemeral public key $R \leftarrow k \cdot G$
    Compute affine coordinates of R
    Compute $s_1 \leftarrow R_x \mod q$
    Compute $s_2 \leftarrow k^{-1}(h + s_1 \cdot x) \mod q$
until $s_1 \neq 0$ and $s_2 \neq 0$
return (s_1, s_2)

Verify. To verify the signature, the verifier computes: $u_1 \equiv h \cdot s_2^{-1} \mod q$ and $u_2 \equiv s_1 \cdot s_2^{-1} \mod q$. Using these values, the verifier computes: $R' \equiv u_1 \cdot G + u_2 \cdot Y$. Since $Y = x \cdot G$, we substitute: $R' \equiv u_1 \cdot G + u_2 \cdot (x \cdot G) = (u_1 + u_2 \cdot x) \cdot G$. Expanding $u_1 + u_2 \cdot x$, $u_1 + u_2 \cdot x = (h \cdot s_2^{-1} + s_1 \cdot s_2^{-1} \cdot x) \mod q$. Factoring out s_2^{-1}, $(h + s_1 \cdot x) \cdot s_2^{-1} \equiv k \cdot s_2^{-1} \mod q$. Thus, we obtain: $R' \equiv k \cdot s_2^{-1} \cdot G = k \cdot G = R$. Since $R' \equiv R$, we conclude: $s_1 = R_x \mod q$.

Algorithm 36 ECElGamal Signature Verification

Input: Message m, Signature (s_1, s_2), Public key Y, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: **Valid** if signature is correct, otherwise **Invalid**

if $s_1 \notin [1, q-1]$ or $s_2 \notin [1, q-1]$ **then**
    return Invalid
end if
Select hash function H from {SHA-2, SHA-3}
Compute hash $h \leftarrow H(m)$
Compute $s_2^{-1} \mod q$
Compute $u_1 \leftarrow h \cdot s_2^{-1} \mod q$
Compute $u_2 \leftarrow s_1 \cdot s_2^{-1} \mod q$
Compute point $R \leftarrow u_1 \cdot G + u_2 \cdot Y$
Compute affine coordinates of R
Compute $v \leftarrow R_x \mod q$
if $v = s_1$ **then**
    return Valid
else
    return Invalid
end if

The interface of its implementation is also similar to the ElGamal cryptosystem, with the main difference is the input type where points on elliptic curve are passed instead of a big number.

```
typedef struct public_params
{
    curve_t curve;
} public_params_t;

typedef struct priv_key
{
    mpz_t x;
} priv_key_t;

typedef struct pub_key
{
    point_t Y;
} pub_key_t;

void setup(public_params_t *pp, ec_t curve_type, unsigned char curve_id);
void keygen(priv_key_t *sk, pub_key_t *pk, const public_params_t *pp);
void crypto_encrypt(mpz_t c1, mpz_t c2, const mpz_t m, const pub_key_t *pk, const public_params_t *pp,
↪  hash_func_t hash_function);
void crypto_decrypt(mpz_t m, const mpz_t c1, const mpz_t c2, const priv_key_t *sk, const public_params_t *pp,
↪  hash_func_t hash_function);
void crypto_sign(mpz_t s1, mpz_t s2, const unsigned char *m, size_t m_len, const priv_key_t *sk, const
↪  public_params_t *pp, hash_func_t hash_function);
char crypto_verify(const mpz_t s1, const mpz_t s2, const unsigned char *m, size_t m_len, const pub_key_t *pk,
↪  const public_params_t *pp, hash_func_t hash_function);
```

The encryption and signing/verification functionalities are tested in separated executable files.

6.3 ECDSA

While the DSA standard has been banned from usage in the latest version of *Digital Signature Standard* [NIS23a], the ECDSA standard still suffices the security requirement of NIST (at least for this decade). It is the elliptic curve analog of DSA. Similar to other modules, we limits the choice of hash functions to `SHA-2` and `SHA-3` families.

Setup. The setup process will initiate the selected elliptic curve.

KeyGen. A secret scalar value is uniformly sampled and used to calculate the corresponding public point on the curve.

Algorithm 37 ECDSA Key Generation

Input: Public parameters $pp = (p, q, G, \text{curve})$
Output: Private key d, Public key Q

Generate random integer d in range $[0, q-1]$
Compute generator point G on the curve
Compute public key $Q \leftarrow d \cdot G$
return (d, Q)

Sign. The signer computes the signature as follows: $r \equiv R_x \mod q$ and $s \equiv k^{-1}(e + d \cdot r) \mod q$.

Algorithm 38 ECDSA Signature Generation

Input: Message m, Private key d, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: Signature (r, s)

Select hash function H
Compute hash $e \leftarrow H(m)$
Adjust e to bit length of q
repeat
    Generate random integer k in range $[0, q-1]$
    Compute ephemeral public key $R \leftarrow k \cdot G$
    Compute affine coordinates of R
    Compute $r \leftarrow R_x \mod q$
    Compute $s \leftarrow k^{-1}(e + d \cdot r) \mod q$
until $r \neq 0$ and $s \neq 0$
return (r, s)

Verify. To verify the signature, the verifier computes: $u \equiv e \cdot s^{-1} \mod q$ and $v \equiv r \cdot s^{-1} \mod q$. Using these values, the verifier computes: $R' \equiv u \cdot G + v \cdot Q$. Since $Q = d \cdot G$, we substitute: $R' \equiv u \cdot G + v \cdot (d \cdot G) = (u + v \cdot d) \cdot G$. Expanding $u + v \cdot d$, $u + v \cdot d = (e \cdot s^{-1} + r \cdot s^{-1} \cdot d) \mod q$. Factoring out s^{-1}, $(e + r \cdot d) \cdot s^{-1} \equiv k \cdot s^{-1} \mod q$. Thus, we obtain: $R' \equiv k \cdot s^{-1} \cdot G = k \cdot G = R$. Since $R' \equiv R$, we conclude: $r = R_x \mod q$.

Algorithm 39 ECDSA Signature Verification

Input: Message m, Signature (r, s), Public key Q, Public parameters $pp = (p, q, G, \text{curve})$, Hash function H
Output: **Valid** if signature is correct, otherwise **Invalid**

if $r \notin [1, q-1]$ or $s \notin [1, q-1]$ **then**
    return Invalid
end if
Select hash function H
Compute hash $e \leftarrow H(m)$
Adjust e to bit length of q
Compute $s^{-1} \mod q$
Compute $u \leftarrow e \cdot s^{-1} \mod q$
Compute $v \leftarrow r \cdot s^{-1} \mod q$
Compute point $R \leftarrow u \cdot G + v \cdot Q$
if R is the identity point **then**
    return Invalid
end if
Compute affine coordinates of R
Compute $r_1 \leftarrow R_x \mod q$
if $r = r_1$ **then**
    return Valid
else
    return Invalid
end if

6.4 ECDH Key Exchange

Similar to the analog of DSA and ECDSA, ECDH is an adaptation of the Diffie-Hellman Key Exchange protocol with the usage of elliptic curves' algebraic group structure. The quadruple of functionalities is always `(Setup, GenSecret, GenPublic, GenSharedSecret)`.

Setup. The setup process will initiate the selected elliptic curve.

GenSecret. Alice and Bob generate a secret scalar within the order of the EC's group.

Algorithm 40 ECDH Secret Generation

Input: Public parameters $pp = (p, q, G, \text{curve})$
Output: Secret key s
1: **Generate** random integer s in range $[0, q-1]$
2: **return** s

GenPublic. The public values are points on the selected curves and can be transformed by serialization/deserialization (conversion between points and bytes).

Algorithm 41 ECDH Public Computation

Input: Secret key s, Public parameters $pp = (p, q, G, \text{curve})$
Output: Public key P
1: **Compute** generator point G on the curve
2: **Compute** public key $P \leftarrow s \cdot G$
3: **return** P

GenSharedSecret. The shared secret is retrieved similarly to DH key exchange.

Algorithm 42 ECDH Shared Secret Computation

Input: Private key s, Other party's public key P, Public parameters $pp = (p, q, G, \text{curve})$
Output: Shared secret K
1: **Compute** $K \leftarrow s \cdot P$
2: **return** K

While also vulnerable to MITM attacks, this elliptic curve version achieves better security with smaller keys, which is the main reason it's favored in most current internet protocols.

7 Cython Wrapper

7.1 Cython Sources

Every module from the C library has its own Cython sources to wrap it into a Python module.

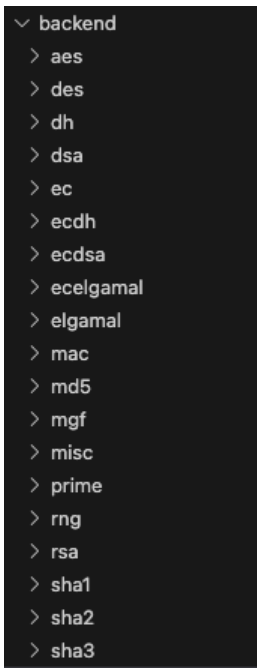


Figure 3: Backend Modules

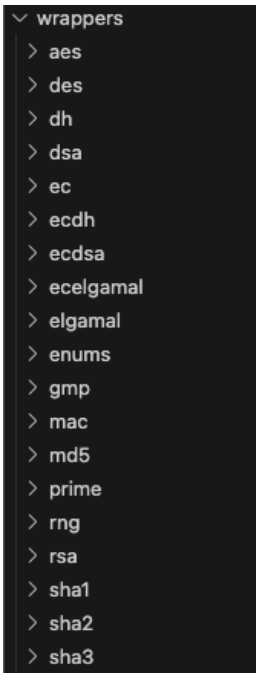


Figure 4: Wrapper Modules

There are two types of Cython source files used in this project: `pxd` and `py` files.

- `pxd` files work like C header files, they contain Cython declarations which are only meant for inclusion by Cython modules. A `.pxd` file is imported into a `.pyx` module by using the `cimport` keyword.
- `.py` files implements Python objects

An example of `pxd` file:

```
from libc.stdlib cimport malloc, free
from libc.stdint cimport uint64_t

cdef extern from "des.h":
    ctypedef enum des_mode_t:
        DES_ENCRYPT,
        DES_DECRYPT

    cdef const size_t _DES_BLOCK_SIZE "DES_BLOCK_SIZE"

    void des(uint64_t *, uint64_t *, const uint64_t, int, des_mode_t)
    void des_file(const char *, const char *, const uint64_t, des_mode_t)

cdef extern from "conversion.h":
    unsigned char *pkcs7_padding(unsigned char *, size_t, size_t *, size_t)
    unsigned char *pkcs7_unpadding(unsigned char *, size_t, size_t *, size_t)

cdef class DES:
    cpdef des(self, bytes data, bytes key, mode)
    cpdef des_file(self, str file_path, str output_path, bytes key, mode)
    cpdef bytes encrypt(self, bytes data, bytes key)
    cpdef bytes decrypt(self, bytes data, bytes key)
    cpdef void encrypt_file(self, str file_path, str output_path, bytes key)
    cpdef void decrypt_file(self, str file_path, str output_path, bytes key)
    cpdef bytes pad(self, bytes data, size_t length)
    cpdef bytes unpad(self, bytes padded_data, size_t length)
```

Resource from the C library can be accessed by using `cdef extern from <header_file>`, such as functions, enum, constant variables, etc.

After the declaration of all necessary resources from the C library, we implement the module's wrapper code in Python.

```python
import cython

@cython.cclass
class DES:
    @cython.ccall
    def des(self, data: bytes, key: bytes, mode):
        assert len(data) / _DES_BLOCK_SIZE == len(data) // _DES_BLOCK_SIZE
        num_blocks = len(data) // _DES_BLOCK_SIZE

        in_blocks = cython.cast(cython.p_ulonglong, malloc(num_blocks * _DES_BLOCK_SIZE))
        out_blocks = cython.cast(cython.p_ulonglong, malloc(num_blocks * _DES_BLOCK_SIZE))

        for i in range(num_blocks):
            in_blocks[i] = cython.cast(cython.ulonglong, int.from_bytes(data[i*8:i*8+8], byteorder='little'))
        key_block = cython.cast(cython.ulonglong, int.from_bytes(key, byteorder='little'))

        des(in_blocks, out_blocks, key_block, num_blocks, mode)
        result = cython.declare(bytes, cython.cast(cython.p_char, out_blocks)[:len(data)])
        free(in_blocks)
        free(out_blocks)
        return result

    @cython.ccall
    def des_file(self, in_path: str, out_path: str, key: bytes, mode):
        infile = in_path.encode('utf-8')
        outfile = out_path.encode('utf-8')

        in_ptr = cython.cast(cython.p_char, infile)
        out_ptr = cython.cast(cython.p_char, outfile)
        key_block = cython.cast(cython.ulonglong, int.from_bytes(key, byteorder='little'))

        des_file(in_ptr, out_ptr, key_block, mode)
```

We can use data types and structures from C with `cast` or `address` functions provided by Cython. We noted that the management of memory needs to be taken into account of, because not like simple Python programs, Cython gives the developers the power to allocate and free memory just like in C.

7.2 Setup Process

After wrapping all the modules, the *cythonization* process can be executed to generate the wrapper library for usage in Python programs.

Each module corresponds to a Python extension, where we can specify the C & Cython source code, the include directories, and extra flags for the compilation. We noted that although it's possible to use the already compiled object files, re-compiling the source file provides more flexibility when we setup the wrapper library, e.g. debugging with address sanitizer, adding special compilation flags, etc.

```python
def get_extension(module):
    name = f'{LIB_NAME}.{module}'
    sources = [f'{WRAPPER_DIR}/{module}/__init__.py']
    objects = []
    if module not in ["enums", "gmp"]:
        objects += get_deps(f'{SRC_DIR}/{module}') + [
            f'{OBJ_DIR}/{module}/{f.replace(".c", ".o")}' for f \
                in os.listdir(os.path.join(SRC_DIR, module)) \
                if f.endswith(".c") and not f.endswith(".test.c")
        ]
    include_dirs = [f'{SRC_DIR}'] + list(set(
        [f'{SRC_DIR}/{"/".join(file.split("/")[1:-1])}' for file in objects]
    ))

```

```
    return Extension(
        name,
        sources=sources,
        extra_objects=objects,
        include_dirs=include_dirs,
        extra_compile_args=CFLAGS,
        extra_link_args=LDFLAGS,
    )
```

We can specify Cython-specific compiler directives for optimization on the generated code. Moreover, the cythonize process can also be parallelized to speed up the development process.

```
class CustomBuildExtension(build_ext):
    def run(self):
        self.build_lib = BUILD_DIR
        self.inplace = False
        self.parallel = int(os.cpu_count() * 1.5)
        super().run()
...
cython_directives = {
    "boundscheck": "False",
    "wraparound": "False",
    "cdivision": "True",
    "language_level": "3",
    "emit_code_comments": "False",
}
...
setup(
    name="wrappers",
    ext_modules=cythonize(
        extensions,
        compiler_directives=cython_directives,
        quiet=True,
        annotate=True,
    ),
    cmdclass={'build_ext': CustomBuildExtension},
    zip_safe=False,
)
```

8 GUI Program

The GUI framework that we utilized is PyQt5, a set of Python bindings for v5 of the Qt application framework from The Qt Company [Com25].

The wireframes are designed with the designer software QtCreator and then loaded into Python programs using the PyQt5 library for the implementation of logic, e.g. buttons' events, I/O events, effects, linkage with Python wrapper modules, etc. In the following sections, we provide the screenshots with a brief introduction how to use the interface.

8.1 Main

The main interface consists of two components: a navigator bar (side bar) on the left and a stack of interfaces on the right. By interacting with the navigator bar, we will basically select an interface to put on top of the stack, which allows users to interact with the cryptographic functionalities of that interface. In our work, the interface *Basic primitives* is set as default for the main interface.

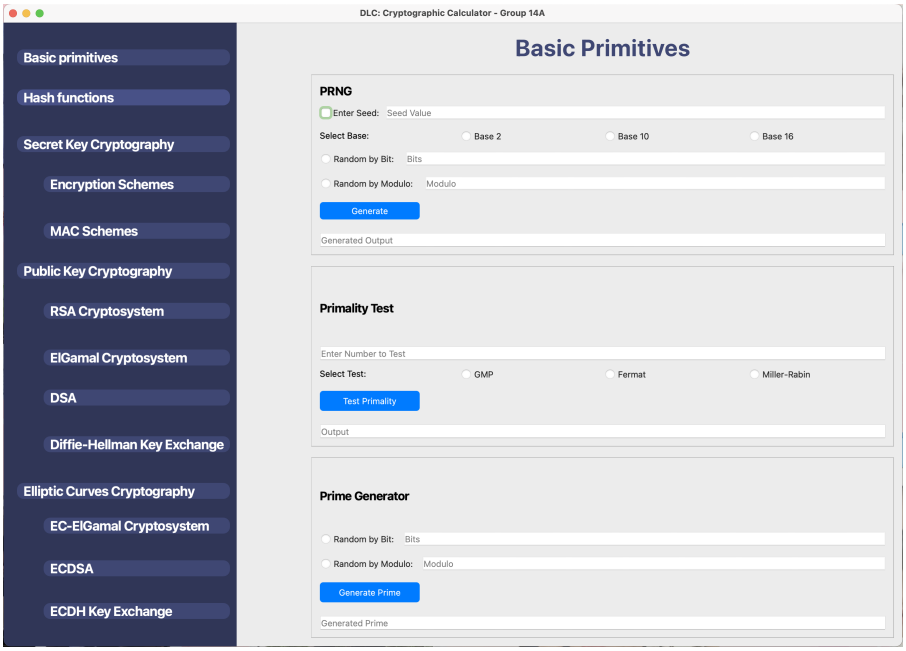


Figure 5: Main UI

8.2 Basic Primitives

This interface supports three types of calculations: Pseudo-Random Number Generation (PRNG), Primality Testing, and Prime Generation.

- In PRNG, users can optionally enter a seed value, select the numerical base, and choose a generation method.
- In Primality Testing, three different test methods are available.
- In Prime Generation, users can generate primes based on either bit length or modulo constraints.

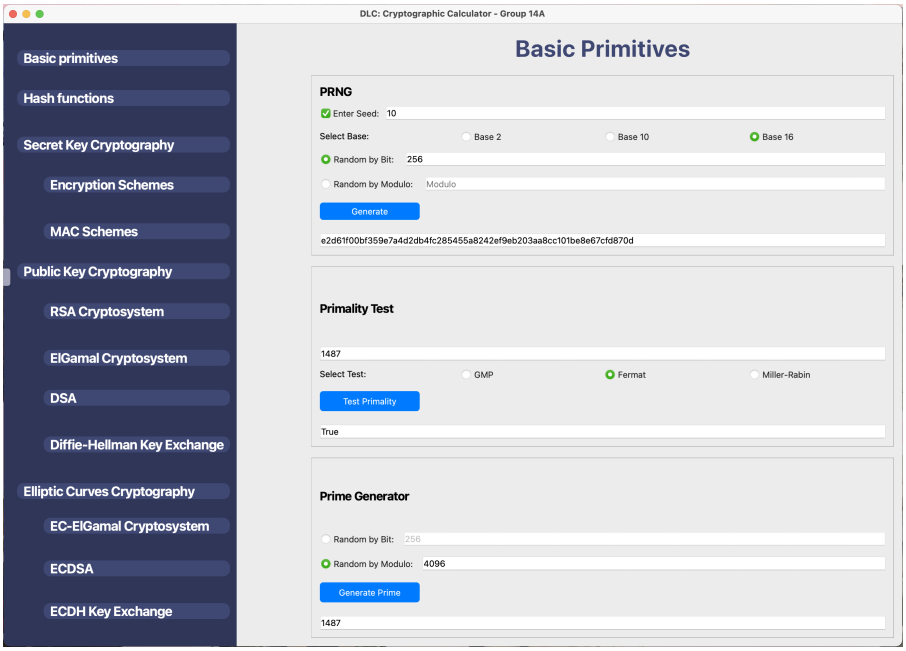


Figure 6: Basic primitives UI

If the input for computation is invalid, an error dialog will appear to notify the user.

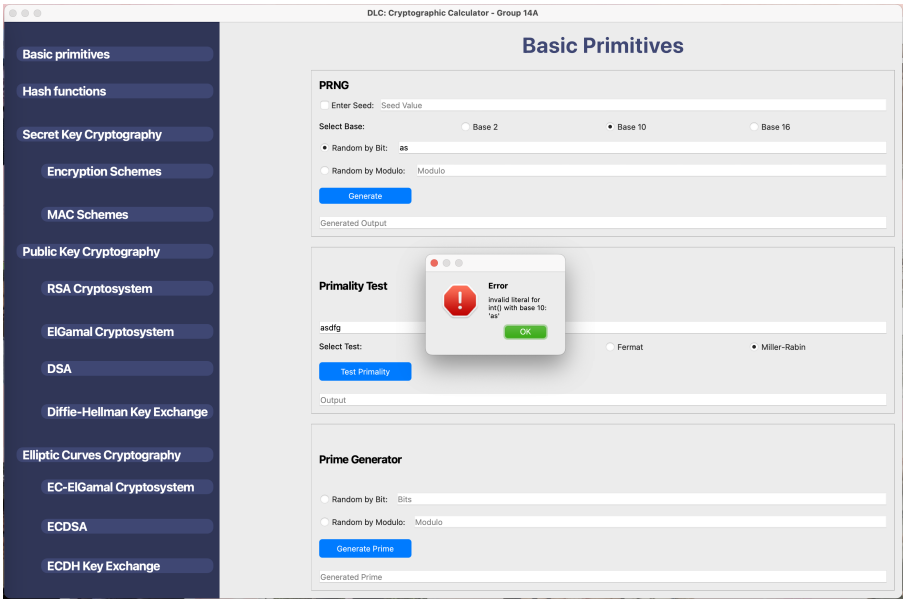
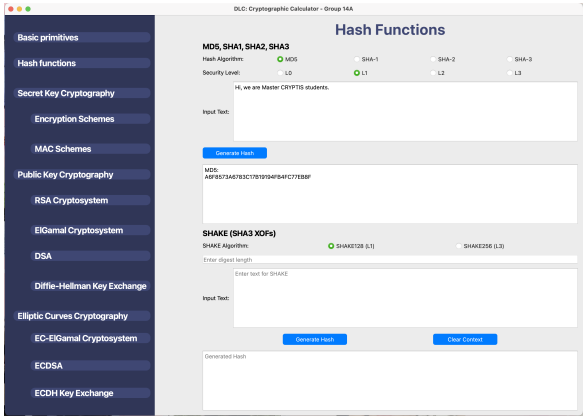


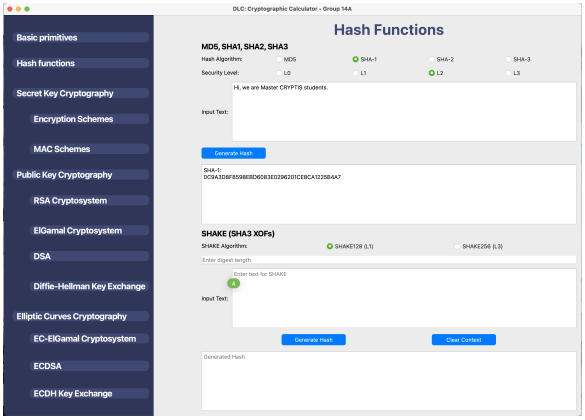
Figure 7: Error UI

8.3 Hash

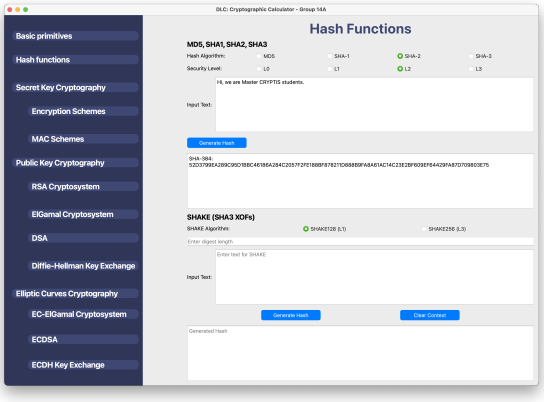
In the *hash* interface, users can choose from a variety of hash algorithms and security levels to meet specific cryptographic needs. The most commonly used hash functions generate outputs of fixed length.



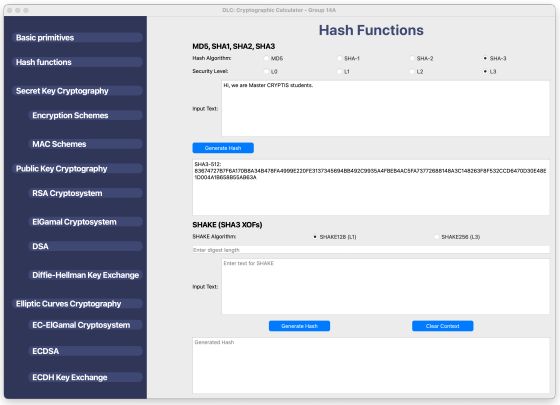
(a) MD5



(b) SHA-1



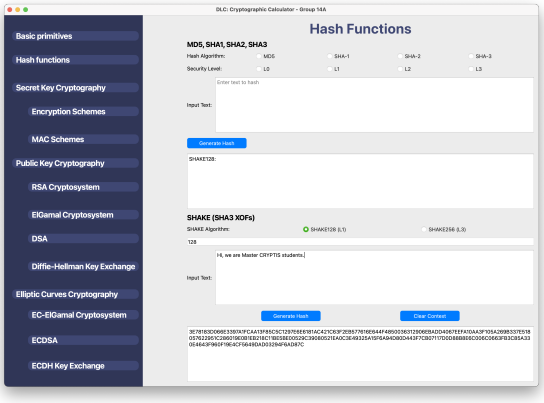
(c) SHA-2



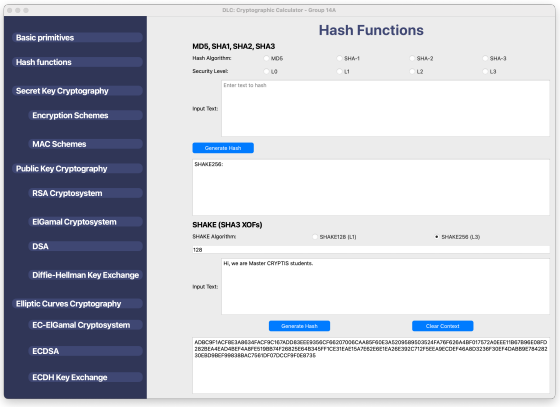
(d) SHA-3

Figure 8: Fixed-length hash functions UI

Moreover, users can also choose the hash function SHAKE for variable-length output.



(a) SHAKE128



(b) SHAKE256

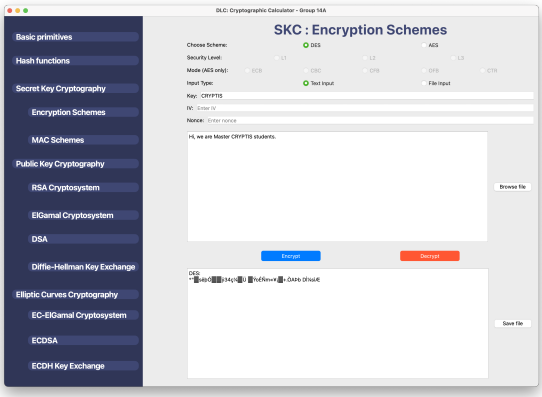
Figure 9: XOF hash functions UI

8.4 Secret Key Cryptography

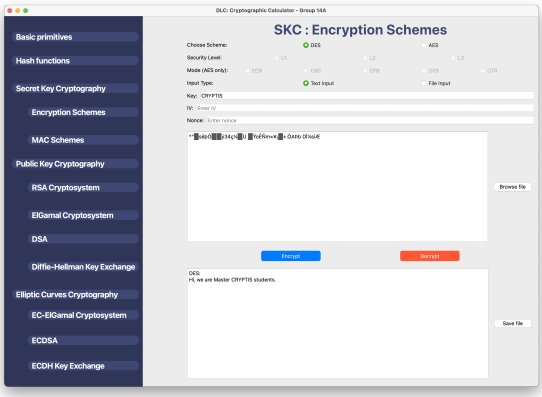
In secret key cryptography, there are two main types of schemes: encryption schemes and message authentication code (MAC) schemes. The input can be either text or a file.

8.4.1 Encryption Schemes

We can select the encryption and decryption algorithm along with the necessary parameters.



(a) Encryption

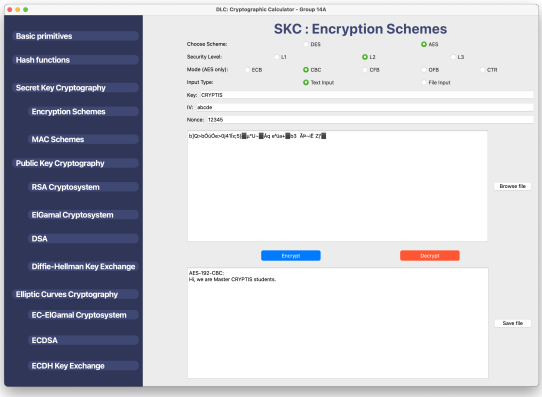


(b) Decryption

Figure 10: DES UI

In comparison with DES, AES will require more configuration options and values.

(a) Encryption

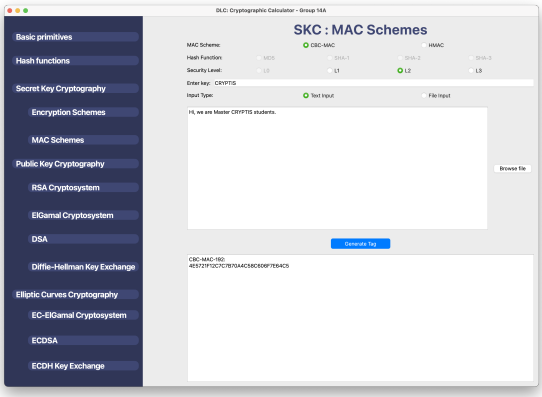


(b) Decryption

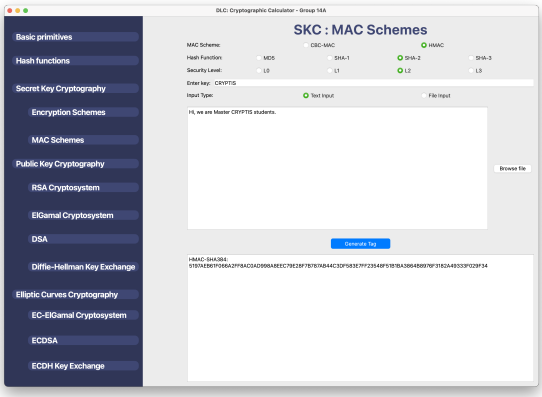
Figure 11: AES UI

8.4.2 MAC Schemes

There are two MAC schemes available: CBC-MAC and HMAC. HMAC requires a hash algorithm, whereas CBC-MAC does not.



(a) CBC-MAC



(b) HMAC

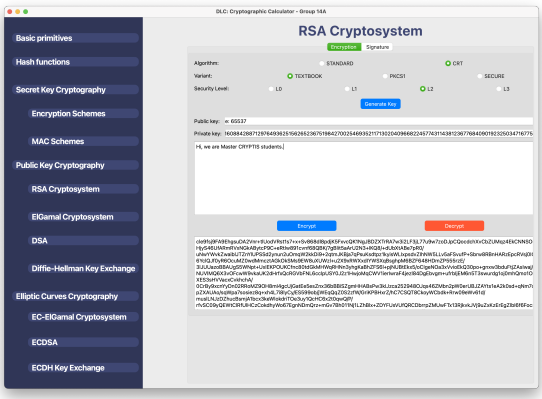
Figure 12: MAC UI

8.5 Public Key Cryptography

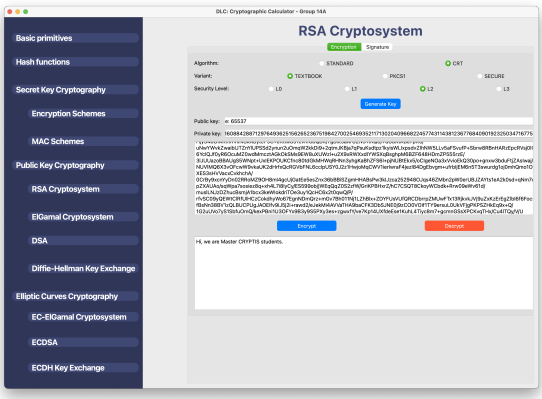
In public key cryptography, users can choose from the RSA cryptosystem, ElGamal cryptosystem, and DSA signature. They can select either an encryption scheme or a signature scheme as needed. A dedicated button is available for generating a key pair or secret (for key exchange). However, the input is restricted to text and does not support files.

8.5.1 RSA Cryptosystem

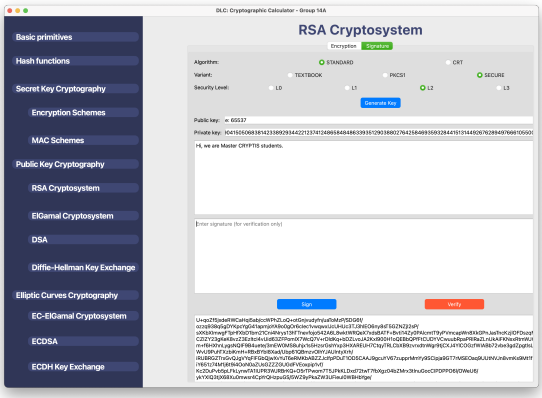
In the RSA cryptosystem, users can select the algorithm, variant, and security level for encryption, decryption, signature generation, or verification.



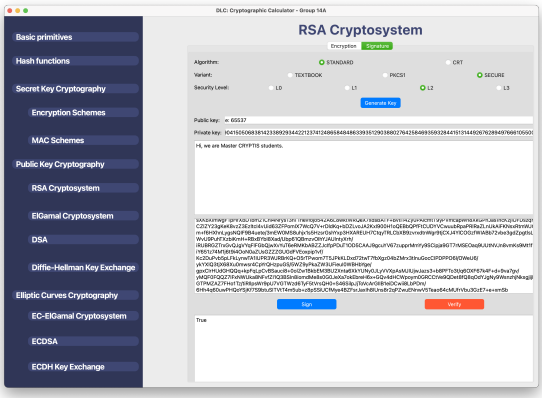
(a) Encryption



(b) Decryption



(c) Signature generation

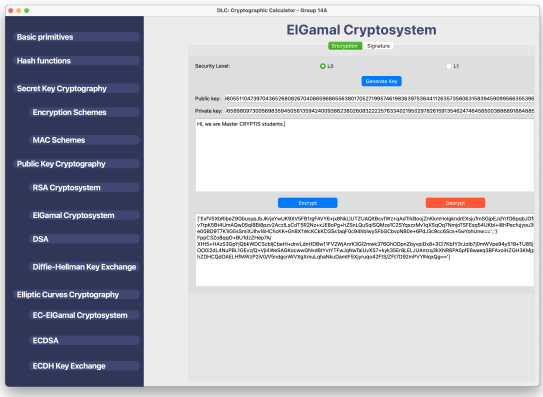


(d) Verification

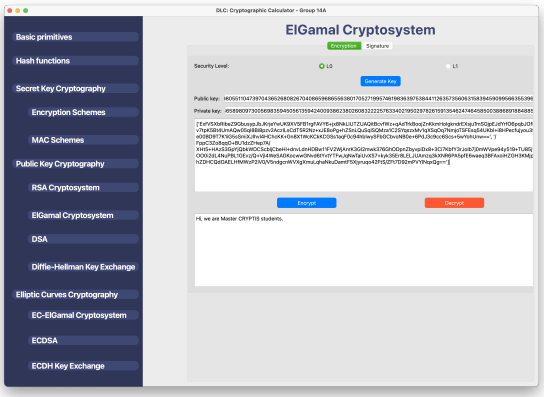
Figure 13: RSA UI

8.5.2 ElGamal Cryptosystem

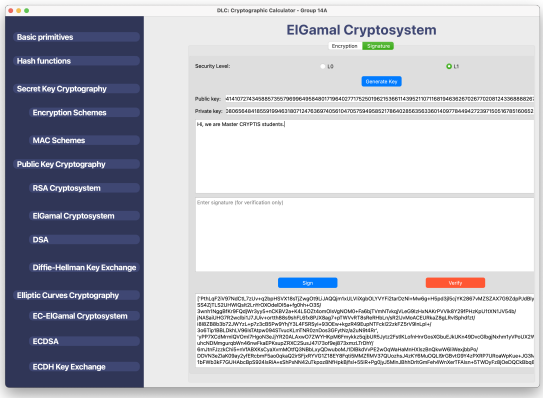
In the ElGamal cryptosystem, users only need to select the security level for encryption, decryption, signature generation, or verification.



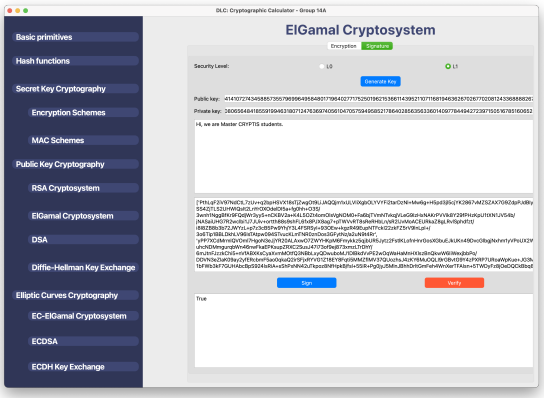
(a) Encryption



(b) Decryption



(c) Signature generation

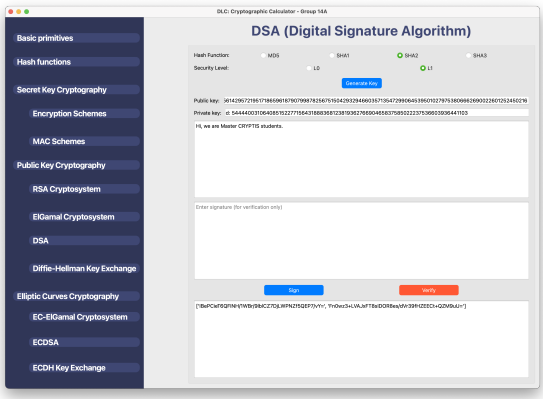


(d) Verification

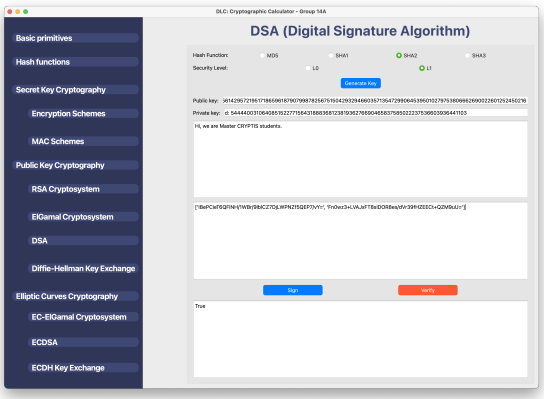
Figure 14: ElGamal UI

8.5.3 DSA Signature

In the Digital Signature Algorithm (DSA), selecting a hash function and specifying the security level are required.



(a) Signature generation



(b) Verification

Figure 15: DSA UI

8.5.4 Diffie-Hellman Key Exchange

To generate a shared secret between A and B, users must select the security level and click the button to generate individual secrets for both A and B.

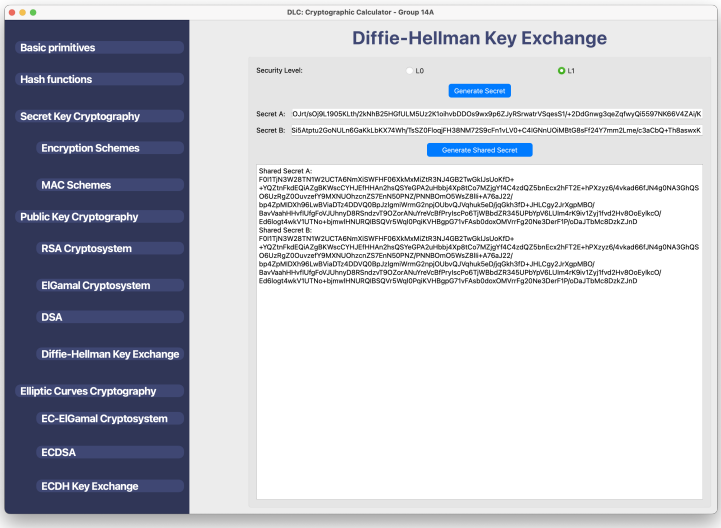


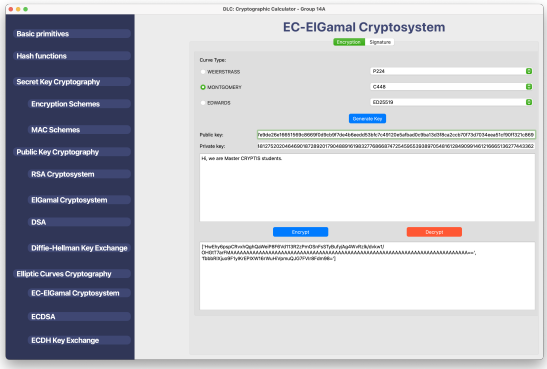
Figure 16: Diffie-Hellman Key Exchange UI

8.6 Elliptic Curves Cryptography

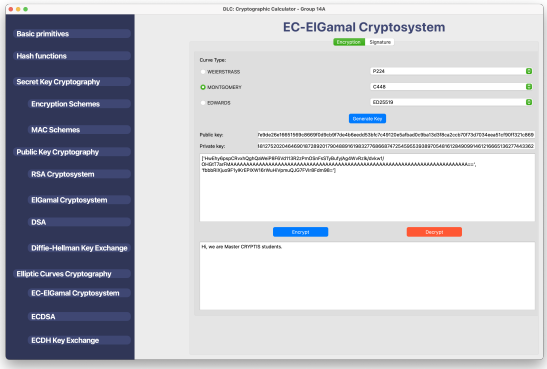
In Elliptic Curve Cryptography, users must select a curve type along with its variant, which serves as the foundation for subsequent calculations. Additionally, a button is provided for generating a key pair or a secret.

8.6.1 EC-ElGamal Cryptosystem

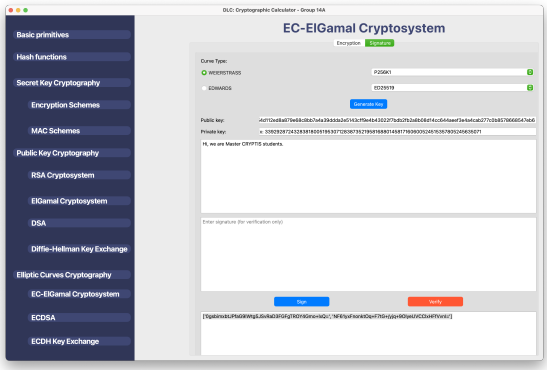
Here are the examples for encryption, decryption, signature generation and verification in EC-ElGamal.



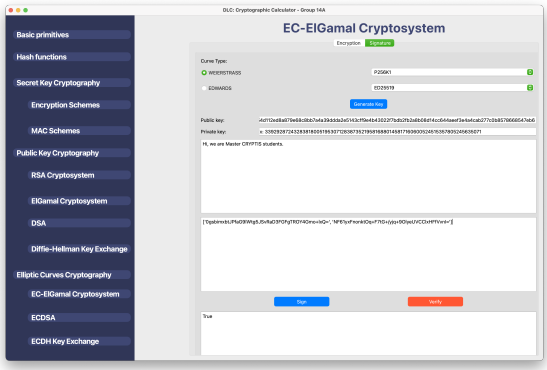
(a) Encryption



(b) Decryption



(c) Signature generation

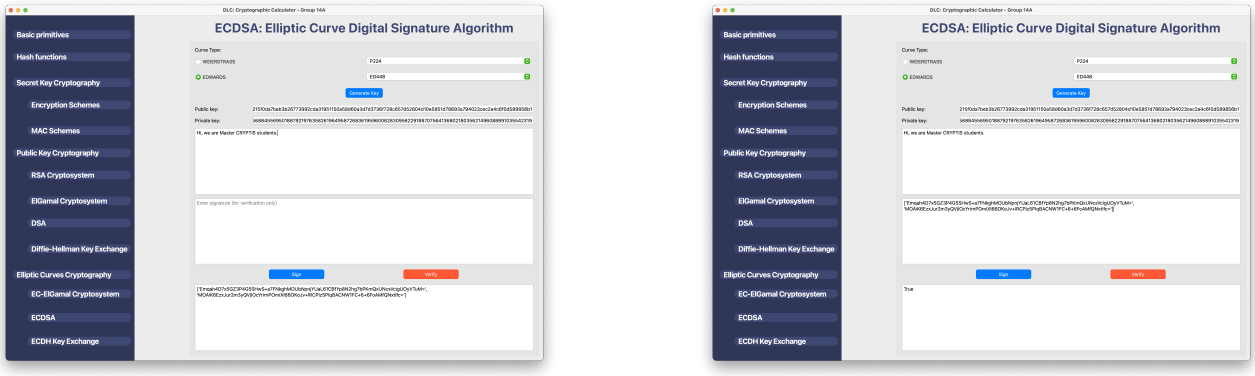


(d) Verification

Figure 17: EC-ElGamal UI

8.6.2 ECDSA Signature

Here are the examples for signature generation, or verification in ECDSA Signature.



(a) Signature

(b) Verification

Figure 18: ECDSA UI

8.6.3 ECDH Key Exchange

After selecting the curve and generate the separate secrets, users can generate the shared secret.

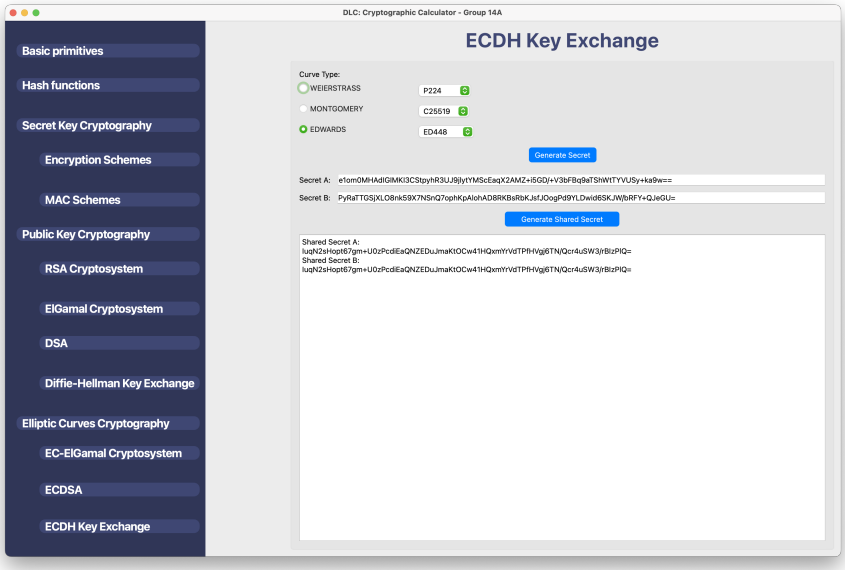


Figure 19: ECDH Key Exchange UI

Conclusion

The respective contributions to our project are outlined as follows.

Name	**Programming**	**Writing**
NDIONGUE Mohamed Lamine	MD5, SHA1, SHA2	Section 3, Appendix B, C, D
NGUYEN Thi Ha Trang	DH, ECDH, Cython Wrapper, GUI	Section 6, 7, 8
NGUYEN Minh Luan	DES, AES, CBC-MAC, Utils	Section 4, Appendix H, I
PHAM Ho Nguyen	PRNG, Prime, SHA3, HMAC, ElGamal, DSA, Elliptic Curves, ECElGamal, ECDSA, Cython Wrapper	Section 1, 2, 5, 6, Appendix A, E, G, J, K
TRAN Xuan Bach	RSA, Utils	Section 5, Appendix F, G

Table 5: Contributions

We reviewed the specified objectives our team laid out initially to see what we have achieved and what has not been delivered.

For the functional requirements, we managed to not only meet the requirements of the project but also implement many more cryptography primitives and protocols:

- ☑ Implementation of a cryptography library in C.
  - ☑ Basic Primitives.
  - ☑ Hash Functions.
  - ☑ Secret Key Cryptography.
  - ☑ Public Key Cryptography.
  - ☑ Elliptic Curve Cryptography.
- ☑ Implementation of a Python wrapper for the C library.
- ☑ Implementation of a GUI program in Python.

Initially, we wanted to implement other advanced primitives such as commitment schemes, pairing-friendly elliptic curves, tree data structures, e.g. Merkle Tree, GGM Tree, secret sharing schemes, etc. but we did not find enough time to complete them. Therefore, this remains a possible direction for any member of the group to continue exploring after the project.

For the non-functional requirements, we achieved the following criteria:

- ☑ Usage of GMP library in core functionalities.
- ☑ Preparation of test vectors and executable test files for all modules in the C library.
- ☑ Following widely adopted standards when it's possible.
- ☑ Support for both text and file inputs in secret key cryptography (encryption schemes and MAC schemes).
- ☑ Support for cross-platform building and testing: Linux, MacOS, and Windows.
- ☑ Sanitized memory management in the C library.

While we discussed some techniques for optimization and protection against side-channel attacks, it also not covered in the scope of this project and remains a potential direction for future work.

References

[AES23] Advanced Encryption Standard (AES). Technical report, 5 2023.

[BBC+11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[BDPA] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak team - keccak and sha-3.

[BL] Daniel J. Bernstein and Tanja Lange. Elliptic curve cryptography – explicit formulas database (efd).

[Com25] Riverbank Computing. Pyqt5 - python bindings for qt, 2025.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[Kal98] B. Kaliski. Rfc2313: Pkcs #1: Rsa encryption version 1.5, 1998.

[KK03] T. Kivinen and M. Kojo. Rfc3526: More modular exponential (modp) diffie-hellman groups for internet key exchange (ike), 2003.

[MKJR16] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Rfc 8017: Pkcs #1: Rsa cryptography specifications version 2.2, 2016.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[NIS13a] NIST. Digital signature standard (dss). Technical Report FIPS PUB 186-4, National Institute of Standards and Technology (NIST), July 2013.

[NIS13b] NIST. Example implementations of the digital signature algorithm (dsa). Technical report, National Institute of Standards and Technology (NIST), 2013.

[NIS15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical Report FIPS PUB 202, National Institute of Standards and Technology (NIST), August 2015.

[NIS23a] NIST. Digital signature standard (dss). Technical Report FIPS PUB 186-5, National Institute of Standards and Technology (NIST), February 2023.

[NIS23b] NIST. Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters. Technical Report NIST SP 800-186, National Institute of Standards and Technology (NIST), February 2023.

[RE24] A.M. Rowsell and Epachamo. File:BlockcipherModesofOperation.png, CC BY-SA 4.0, 2024. [Online; accessed 03-February-2025].

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

A Building & Debugging

Building with Make

Our library enables cross-platform (Linux, MacOS, Windows) building and testing by utilizing Make. Users are recommended to follow the tutorial in `README.md` to build and test the library as well as the GUI program.

```
all test prepare:
	@cd src/backend && make $@

setup:
	./setup.sh

build_fast:
	@cd src/backend && make all -j $(NUM_CORES)

test_%: prepare
	@cd $(SRC_DIR)/$* && make test

clean: clean_wrapper
	@cd src/backend && make $@

clean_make:
	@rm -rf $(SRC_DIR)/*/Makefile

clean_wrapper:
	@rm -rf ./**/*cpython-*
	@rm -rf build/wrappers
	@find . -type f -path "*/wrappers/*.c*" -exec rm {} \;

wrap: all
	@python3 src/backend/wrappers/setup.py build_ext
	@python3 src/backend/wrappers/setup.py clean

gui: wrap
	@python3 src/frontend/main.py

gui_%:
	@python3 src/frontend/$*.py
```

While users are recommended to interact with the global `Makefile`, they can go build and test each module in the C library independently.

Moreover, if users have access to the number of available processors in their environment, they can optimize the building process by parallelization.

Debugging with compiler's sanitizers

Normally, the library will be built and wrapped in production mode. If the debug mode is specified in the environment file `.env`, users can debug their programs using different types of sanitizer available, depending on the compiler and the host environment. Our implementation of the C library and the Cython wrapper has already been sanitized by AddressSanitizer, LeakSanitizer, and UndefinedSanitizer, which detected many mistakes we made, e.g. writing outside of allocated memory, allocated memory not freed, use-after-free mistakes, unexpected undefined behavior, etc.

```
# App Environments: production,development,debug
APP_ENV=debug
...
# Sanitizers: address,leak,thread,undefined
SANITIZER=address
SAN_CFLAGS=-fsanitize=$(SANITIZER) -fsanitize-recover=$(SANITIZER) -fno-omit-frame-pointer -g
SAN_LDFLAGS=-fsanitize=$(SANITIZER) -fsanitize-recover=$(SANITIZER) -shared-libsan
```

B MD5 Implementation Details

The steps of the function are as follows:

MD5.c

```
    #include "md5.h"

#define ROTLEFT(a, b) ((a << b) | (a >> (32 - b))) // rotate left
#define F(x, y, z) ((x & y) | (~x & z))            // F function
#define G(x, y, z) ((x & z) | (y & ~z))            // G function
#define H(x, y, z) (x ^ y ^ z)                     // H function
#define I(x, y, z) (y ^ (x | ~z))                  // I function
```

The line #**include "md5.h"** defines the inclusion of the md5.h file. Then, we have the definition of some functions defined as constants in the #**define** section including:

- $ROTLEFT(a, b)((a << b)|(a >> (32 - b)))$: defined as a constant for left rotation, that is, circular shifting of the bits of an integer a to the left by b positions $(a << b)$ and $(a >> (32 - b))$ allows shifting the bits of a to the right by $32 - b$ positions with a concatenation operator | combining the two results.
- The following logical functions used in the MD5 cryptographic hash algorithm allow non-linear transformations of the bits to ensure that the results are difficult to invert and predict. In other words, the operations performed do not follow a direct or simple relationship.
  - $F(x, y, z)((x\&y)|(\sim x\&z))$: This function defined in the #define section uses a combination of logical AND (&), OR (|), and NOT ($\sim$) operators and returns the bits of y if $x = 1$ and the bits of z if $x = 0$.
  - $G(x, y, z)((x \sim z)|(y \sim z))$: Similarly to the previous one, it defines a combination of logical operations between AND (&), OR (|), and NOT ($\sim$) and returns the bits of x if $z = 1$ and y if $z = 0$.
  - $H(x, y, z)(x \wedge y \wedge z)$: This one performs an XOR operation ($\wedge$) on the three variables x, y, and z and returns 1 if the number of bits set to 1 is odd.
  - $I(x, y, z)(y \wedge (x| \sim z))$: This operation first performs an XOR operation (|) on x and the complement ($\sim$) of z, then an XOR operation on y.

    Then, we have other functions defined in the #define section that will be used to define the md5_transform function. Among them, we have:

```
    #define FF(a, b, c, d, x, s, ac)  \
    {                                 \
        a += F(b, c, d) + x + ac; \
        a = ROTLEFT(a, s);        \
        a += b;                   \
    } // FF function which calls F function and rotates left  and adds a and b
```

This macro function performs a series of logical operations a += F(b, c, d) + x + ac, a = ROTLEFT(a, s) and a += b defined from the previous macros F and ROTLEFT.

```
        #define GG(a, b, c, d, x, s, ac)  \
    {                                 \
        a += G(b, c, d) + x + ac; \
        a = ROTLEFT(a, s);        \
        a += b;                   \
    } // GG function which calls G function and rotates left  and adds a and b.
```

Similar to the previous one, except that G is used instead of F.

```
    #define HH(a, b, c, d, x, s, ac)  \
    {                                 \
        a += H(b, c, d) + x + ac; \
        a = ROTLEFT(a, s);        \
        a += b;                   \
```

```
    } // HH function which calls H function and rotates left  and adds a and b.
```

```
    #define II(a, b, c, d, x, s, ac)  \
    {                                 \
        a += I(b, c, d) + x + ac; \
        a = ROTLEFT(a, s);        \
        a += b;                   \
    } // II function which calls I function and rotates left  and adds a and b.
```

md5_transform

```
    void md5_transform(md5_ctx *ctx, const BYTE data[])
{
    uint32_t a, b, c, d, m[16], i, j;
    for (i = 0, j = 0; i < 16; ++i, j += 4)
    {
        m[i] = (data[j]) + (data[j + 1] << 8) + (data[j + 2] << 16) + (data[j + 3] << 24);
    }
    a = ctx->state[0];
    b = ctx->state[1];
    c = ctx->state[2];
    d = ctx->state[3];

    FF(a, b, c, d, m[0], 7, 0xd76aa478);
    FF(d, a, b, c, m[1], 12, 0xe8c7b756);
    FF(c, d, a, b, m[2], 17, 0x242070db);
    FF(b, c, d, a, m[3], 22, 0xc1bdceee);
    FF(a, b, c, d, m[4], 7, 0xf57c0faf);
    FF(d, a, b, c, m[5], 12, 0x4787c62a);
    FF(c, d, a, b, m[6], 17, 0xa8304613);
    FF(b, c, d, a, m[7], 22, 0xfd469501);
    FF(a, b, c, d, m[8], 7, 0x698098d8);
    FF(d, a, b, c, m[9], 12, 0x8b44f7af);
    FF(c, d, a, b, m[10], 17, 0xffff5bb1);
    FF(b, c, d, a, m[11], 22, 0x895cd7be);
    FF(a, b, c, d, m[12], 7, 0x6b901122);
    FF(d, a, b, c, m[13], 12, 0xfd987193);
    FF(c, d, a, b, m[14], 17, 0xa679438e);
    FF(b, c, d, a, m[15], 22, 0x49b40821);

    GG(a, b, c, d, m[1], 5, 0xf61e2562);
    GG(d, a, b, c, m[6], 9, 0xc040b340);
    GG(c, d, a, b, m[11], 14, 0x265e5a51);
    GG(b, c, d, a, m[0], 20, 0xe9b6c7aa);
    GG(a, b, c, d, m[5], 5, 0xd62f105d);
    GG(d, a, b, c, m[10], 9, 0x02441453);
    GG(c, d, a, b, m[15], 14, 0xd8a1e681);
    GG(b, c, d, a, m[4], 20, 0xe7d3fbc8);
    GG(a, b, c, d, m[9], 5, 0x21e1cde6);
    GG(d, a, b, c, m[14], 9, 0xc33707d6);
    GG(c, d, a, b, m[3], 14, 0xf4d50d87);
    GG(b, c, d, a, m[8], 20, 0x455a14ed);
    GG(a, b, c, d, m[13], 5, 0xa9e3e905);
    GG(d, a, b, c, m[2], 9, 0xfcefa3f8);
    GG(c, d, a, b, m[7], 14, 0x676f02d9);
    GG(b, c, d, a, m[12], 20, 0x8d2a4c8a);

    HH(a, b, c, d, m[5], 4, 0xfffa3942);
    HH(d, a, b, c, m[8], 11, 0x8771f681);
    HH(c, d, a, b, m[11], 16, 0x6d9d6122);
    HH(b, c, d, a, m[14], 23, 0xfde5380c);
    HH(a, b, c, d, m[1], 4, 0xa4beea44);
    HH(d, a, b, c, m[4], 11, 0x4bdecfa9);
    HH(c, d, a, b, m[7], 16, 0xf6bb4b60);
    HH(b, c, d, a, m[10], 23, 0xbebfbc70);
    HH(a, b, c, d, m[13], 4, 0x289b7ec6);
```

```
    HH(d, a, b, c, m[0], 11, 0xeaa127fa);
    HH(c, d, a, b, m[3], 16, 0xd4ef3085);
    HH(b, c, d, a, m[6], 23, 0x04881d05);
    HH(a, b, c, d, m[9], 4, 0xd9d4d039);
    HH(d, a, b, c, m[12], 11, 0xe6db99e5);
    HH(c, d, a, b, m[15], 16, 0x1fa27cf8);
    HH(b, c, d, a, m[2], 23, 0xc4ac5665);

    II(a, b, c, d, m[0], 6, 0xf4292244);
    II(d, a, b, c, m[7], 10, 0x432aff97);
    II(c, d, a, b, m[14], 15, 0xab9423a7);
    II(b, c, d, a, m[5], 21, 0xfc93a039);
    II(a, b, c, d, m[12], 6, 0x655b59c3);
    II(d, a, b, c, m[3], 10, 0x8f0ccc92);
    II(c, d, a, b, m[10], 15, 0xffeff47d);
    II(b, c, d, a, m[1], 21, 0x85845dd1);
    II(a, b, c, d, m[8], 6, 0x6fa87e4f);
    II(d, a, b, c, m[15], 10, 0xfe2ce6e0);
    II(c, d, a, b, m[6], 15, 0xa3014314);
    II(b, c, d, a, m[13], 21, 0x4e0811a1);
    II(a, b, c, d, m[4], 6, 0xf7537e82);
    II(d, a, b, c, m[11], 10, 0xbd3af235);
    II(c, d, a, b, m[2], 15, 0x2ad7d2bb);
    II(b, c, d, a, m[9], 21, 0xeb86d391);

    ctx->state[0] += a;
    ctx->state[1] += b;
    ctx->state[2] += c;
    ctx->state[3] += d;
}
```

In this function md5_transform(md5_ctx *ctx, const BYTE data[]), the various macros defined in the #define section (FF, GG, HH, and II) and a pointer type ctx associated with the state array of size 4 are used to determine the variables a, b, c, and d as well as a loop to determine the values of the m array of size 16 (m[i] = (data[j]) + (data[j + 1] « 8) + (data[j + 2] « 16) + (data[j + 3] « 24)). The hexadecimal values defined in the functions FF, GG, HH, and II of this function are specific constants used in the MD5 algorithm. These constants are derived from the fractional parts of the square roots of prime numbers. They are used to add complexity and non-linearity to the hashing algorithm. They are used in the various transformation steps to mix the bits in a complex and secure way.

md5_init:

```
oid md5\_init(md5\_ctx *ctx)
{
    ctx->datalen = 0;
    ctx->bitlen = 0;
    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xefcdab89;
    ctx->state[2] = 0x98badcfe;
    ctx->state[3] = 0x10325476;
}
```

It is an initialization function for the datalen, bitlen variables, and the state array of size 4: ctx→datalen is initialized to 0, ctx->bitlen is initialized to 0, ctx→state[0] is initialized to 0x67452301, ctx→state[1] is initialized to 0xefcdab89, ctx→state[2] initialized to 0x98badcfe, and ctx→state[3] initialized to 0x10325476.

md5_update:

```
void md5_update(md5_ctx *ctx, const BYTE data[], size_t len)
{
    uint32_t i;
    for (i = 0; i < len; ++i)
    {
```

```
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        if (ctx->datalen == 64)
        {
            md5_transform(ctx, ctx->data);
            ctx->bitlen += 512;
            ctx->datalen = 0;
        }
    }
}
```

This md5_update function creates a for loop incremented to a maximum size len in which the data array of index ctx→datalen is assigned to the data array of index i from the for loop. Then, the datalen index is incremented and after the condition **if(ctx→datalen == 64)** in which the previous **md5_transform** function is called to calculate the bitlen value incremented by 512 bits and datalen of value 0 if true.

md5_final(md5_ctx *ctx, BYTE hash[])

```
void md5_final(md5_ctx *ctx, BYTE hash[])
{
    uint32_t i = ctx->datalen;
    if (ctx->datalen < 56)
    {
        ctx->data[i++] = 0x80;
        while (i < 56)
        {
            ctx->data[i++] = 0x00;
        }
    }
    else
    {
        ctx->data[i++] = 0x80;
        while (i < 64)
        {
            ctx->data[i++] = 0x00;
        }
        md5_transform(ctx, ctx->data);
        memset(ctx->data, 0, 56);
    }
    ctx->bitlen += ctx->datalen * 8;
    ctx->data[56] = ctx->bitlen;
    ctx->data[57] = ctx->bitlen >> 8;
    ctx->data[58] = ctx->bitlen >> 16;
    ctx->data[59] = ctx->bitlen >> 24;
    ctx->data[60] = ctx->bitlen >> 32;
    ctx->data[61] = ctx->bitlen >> 40;
    ctx->data[62] = ctx->bitlen >> 48;
    ctx->data[63] = ctx->bitlen >> 56;
    md5_transform(ctx, ctx->data);
    for (i = 0; i < 4; ++i)
    {
        hash[i] = (ctx->state[0] >> (i * 8)) & 0x000000ff;
        hash[i + 4] = (ctx->state[1] >> (i * 8)) & 0x000000ff;
        hash[i + 8] = (ctx->state[2] >> (i * 8)) & 0x000000ff;
        hash[i + 12] = (ctx->state[3] >> (i * 8)) & 0x000000ff;
    }
}
```

*md5(const void *m, size_t len, void *md, size_t md_len)

```
void *md5(const void *m, size_t len, void *md, size_t md_len)
{
    UNUSED(md_len);
    md5_ctx ctx;
    md5_init(&ctx);
```

```
    md5_update(&ctx, (BYTE *)m, len);
    md5_final(&ctx, (BYTE *)md);
    return md;
}
```

md5.test.c

In this file, we have the definition of the md5_test function, which takes no arguments and returns the integer value **pass**. In this function, we have arrays text1, text2, text3_1, text3_2, hash1, hash2, hash3, buf of size 16 in BYTE (BYTE is a type), where text1 is initialized to an empty character ({""}), text2 to the string "abc" ({"abc"}), text3_1 to an alphabetic character string in both uppercase and lowercase ({"ABCDEFGHIJKLMNOPQRSTUVWXYZabcde"}), and text3_2 to an alphanumeric character string ({"fghijklmnopqrstuvwxyz0123456789"}). Then, various functions (md5_init, md5_update, and md5_final) are called, and at the end of operations, a comparison is made using the memcmp comparison function (defined in the string.h library) with arguments for different hashes, buf, and MD5_BLOCK_SIZE, and the pass variable using the logical && operator.

Finally, we have the main function without any arguments, which tests the md5_test function with an if condition and then displays the result of the condition: if true, we get the result "passed"; otherwise, we get the result "Failed".

```
#include "md5.h"

int md5_test() {
    BYTE text1[] = {""};
    BYTE text2[] = {"abc"};
    BYTE text3_1[] = {"ABCDEFGHIJKLMNOPQRSTUVWXYZabcde"};
    BYTE text3_2[] = {"fghijklmnopqrstuvwxyz0123456789"};
    BYTE hash1[MD5_BLOCK_SIZE] = {0xd4, 0x1d, 0x8c, 0xd9, 0x8f, 0x00, 0xb2, 0x04, 0xe9, 0x80, 0x09, 0x98,
    ↪  0xec, 0xf8, 0x42, 0x7e};
    BYTE hash2[MD5_BLOCK_SIZE] = {0x90, 0x01, 0x50, 0x98, 0x3c, 0xd2, 0x4f, 0xb0, 0xd6, 0x96, 0x3f, 0x7d,
    ↪  0x28, 0xe1, 0x7f, 0x72};
    BYTE hash3[MD5_BLOCK_SIZE] = {0xd1, 0x74, 0xab, 0x98, 0xd2, 0x77, 0xd9, 0xf5, 0xa5, 0x61, 0x1c, 0x2c,
    ↪  0x9f, 0x41, 0x9d, 0x9f};
    BYTE buf[16];
    md5_ctx ctx;
    int pass = 1;

    md5_init(&ctx);
    md5_update(&ctx, text1, strlen((char const*)text1));
    md5_final(&ctx, buf);
    pass = pass && !memcmp(hash1, buf, MD5_BLOCK_SIZE);

    md5_init(&ctx);
    md5_update(&ctx, text2, strlen((char const*)text2));
    md5_final(&ctx, buf);
    pass = pass && !memcmp(hash2, buf, MD5_BLOCK_SIZE);

    md5_init(&ctx);
    md5_update(&ctx, text3_1, strlen((char const*)text3_1));
    md5_update(&ctx, text3_2, strlen((char const*)text3_2));
    md5_final(&ctx, buf);
    pass = pass && !memcmp(hash3, buf, MD5_BLOCK_SIZE);

    return pass;
}

int main() {
    printf("\n================ MD5 Test ================\n");
    if (md5_test()) {
        printf("MD5 Test Passed\n");
    } else {
        printf("MD5 Test Failed\n");
    }
    return 0;
```

```
42 }
```

C SHA-1 Implementation Details

void sha1_transform(sha1_ctx *ctx, const BYTE data[])

Next, we have the definition of the **sha1_transform** function, which performs the main transformation on a 512-bit data block and consists of two parameters: one pointer type and the other a variable named data of type **const BYTE**. In this function, 4 loops are defined:

- In this loop, the m array of size 16 words is assigned a combination term of 4 successive bytes from the data array into a single 32-bit word, described as follows.

```
for (i = 0, j = 0; i < 16; ++i, j += 4)
        m[i] = (data[j] << 24) + (data[j + 1] << 16) + (data[j + 2] << 8) + (data[j + 3]);
```

  - data[j] $\ll$ 24: Shifts the bits of data[j] 24 positions to the left, placing this byte in the most significant bits of the 32-bit integer.
  - data[j + 1] $\ll$ 16: Shifts the bits of data[j + 1] 16 positions to the left, placing this byte in the next bits of the 32-bit integer.
  - data[j + 2] $\ll$ 8: Shifts the bits of data[j + 2] 8 positions to the left, placing this byte in the next bits of the 32-bit integer.
  - data[j + 3]: Places the bits of data[j + 3] in the least significant bits of the 32-bit integer. The results of these shifts are added together to form a 32-bit integer, then stored in the m array.

    Example:

```
data = {0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0, ...}
First iteration: i = 0, j = 0
m[0] = (0x12 << 24) + (0x34 << 16) + (0x56 << 8) + 0x78
     = 0x12000000 + 0x00340000 + 0x00005600 + 0x00000078
     = 0x12345678

Second iteration: i = 1, j = 4
m[1] = (0x9A << 24) + (0xBC << 16) + (0xDE << 8) + 0xF0
     = 0x9A000000 + 0x00BC0000 + 0x0000DE00 + 0x000000F0
     = 0x9ABCDEF0

And so on for the first 16 words of m.
```

- The second loop:

```
for (; i < 80; ++i){
    m[i] = (m[i - 3] ^ m[i - 8] ^ m[i - 14] ^ m[i - 16]);
    m[i] = (m[i] >> 1) | (m[i] >> 31);
}
```

  - With an $m[80]$ array of size 80, this loop performs a XOR ($\hat{}$) operation between the $m[i-j]$ elements with $j \in \{3, 8, 14, 16\}$ assigned to the $m[i]$ element of the $m[80]$ array and another logical operation allowing $m[i]$ to receive the concatenation of $m[i] \ll 1$ (left shift by one position) and $m[i] \gg 31$ (right shift by 31 positions).

    In other words, this code snippet defined in this loop continues preparing the m array for the additional 64 words needed for the SHA-1 algorithm. The line m[i] = (m[i - 3] $\wedge$ m[i - 8] $\wedge$ m[i - 14] $\wedge$ m[i - 16]) calculates the value of m[i] using a combination XOR ($\wedge$) of the corresponding terms with an operator. Then, we have this line of code m[i] = (m[i] $\ll$ 1) | (m[i] $\gg$ 31) performing a logical calculation of a concatenation operator between two shift operations as follows:

    - $\star$ m[i] $\ll$ 1: bit shift of m[i] one position to the left.
    - $\star$ m[i] $\gg$ 31: bit shift of m[i] 31 positions to the right.

- In this code, we have the initialization of the variables a, b, c, d, and e defined as `uint32_t`, described as follows.

```
a = ctx->state[0];
b = ctx->state[1];
c = ctx->state[2];
d = ctx->state[3];
e = ctx->state[4];
```

- ⋆ a = ctx->state[0]: Initialization of a with the first value of the context state (ctx).
- ⋆ b = ctx->state[1]: Initialization of b with the second value of the context state.
- ⋆ c = ctx->state[2]: Initialization of c with the third value of the context state.
- ⋆ d = ctx->state[3]: Initialization of d with the fourth value of the context state.
- ⋆ d = ctx->state[4]: Initialization of e with the fifth value of the context state.

The context state (ctx->state defined in the sha1.h structure type) is an array of five 32-bit integers that maintain the intermediate hash state. These values are initialized in the sha1_init function and updated after each data block transformation.

```
for (i = 0; i < 20; ++i){
    t = ROTLEFT(a, 5) + ((b & c) ^ (~ b & d)) + e + ctx->k[0] + m[i];
    e = d;
    d = c;
    c = ROTLEFT(b, 30);
    b = a;
    a = t;
}
```

```
for (; i < 40; ++i){
    t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[1] + m[i];
    e = d;
    d = c;
    c = ROTLEFT(b, 30);
    b = a;
    a = t;
}
```

```
for (; i < 60; ++i){
    t = ROTLEFT(a, 5) + ((b & c) ^ (b & d) ^ (c & d)) + e + ctx->k[2] + m[i];
    e = d;
    d = c;
    c = ROTLEFT(b, 30);
    b = a;
    a = t;
}
```

```
for (; i < 80; ++i){
    t = ROTLEFT(a, 5) + (b ^ c ^ d) + e + ctx->k[3] + m[i];
    e = d;
    d = c;
    c = ROTLEFT(b, 30);
    b = a;
    a = t;
}
```

These four previous for loops of 20, 40, 60, and 80 rounds in the SHA-1 transformation are used to update each variable a, b, c, d, and e defined earlier and perform the following steps:

- ⋆ t = ROTLEFT(a, 5) + ((b & c) ∧ (∼b & d)) + e + ctx->k[0] + m[i] (for the 20-round for loop): This performs a set of operations: a ROTLEFT(a, 5) left rotation of 5 bits on a, calculates a non-linear function (b & c) ∧ (∼b & d) using variables b, c, and d.
- ⋆ t = ROTLEFT(a, 5) + (b ∧ c ∧ d) + e + ctx->k[1] + m[i] (for the 40-round for loop): This line of code performs a new calculation for the t variable using a combination of ROTLEFT(a,5) to the left, XOR, and an addition of constants and message words. With this for loop, the operations are performed from 20 to 39.

- ⋆ t = ROTLEFT(a, 5) + ((b & c) ∧ (b & d) ∧ (c & d)) + e + ctx->k[2] + m[i] (for the 60-round for loop): Similarly to the previous one, this line of code updates the new value of the t variable. Thus, t is assigned a term composed of ROTLEFT(a,5) for left rotation combined with a logical calculation of boolean operations of AND (&) and XOR (∧) and an addition of the e variable and an addition of a round constant k[2] plus the value of the word m[i] from the specific message from rounds 40 to 59.
- ⋆ t = ROTLEFT(a, 5) + (b ∧ c ∧ d) + e + ctx->k[3] + m[i] (for the 80-round for loop): Similar to the previous ones, but with a round constant addition ctx->k[3], where the for loop would perform its operations from 60 to 79.

  The following operations are similar in the four for loops.
- ⋆ e = d: Updates the e variable to receive d.
- ⋆ d = c: Updates the d variable to receive c.
- ⋆ c = ROTLEFT(b, 30): Updates the c variable with the value of b after a left rotation of 30 bits.
- ⋆ b = a: Updates the b variable to receive a.
- ⋆ a = t: Updates the a variable to receive t.

void sha1_init(sha1_ctx *ctx)

```
 void sha1_init(sha1_ctx *ctx)
{
    ctx->datalen = 0;
    ctx->bitlen = 0;
    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xEFCDAB89;
    ctx->state[2] = 0x98BADCFE;
    ctx->state[3] = 0x10325476;
    ctx->state[4] = 0xc3d2e1f0;
    ctx->k[0] = 0x5a827999;
    ctx->k[1] = 0x6ed9eba1;
    ctx->k[2] = 0x8f1bbcdc;
    ctx->k[3] = 0xca62c1d6;
}
```

void sha1_update(sha1_ctx *ctx, const BYTE data[], size_t len)

```
void sha1_update(sha1_ctx *ctx, const BYTE data[], size_t len)
{
    size_t i;
    for (i = 0; i < len; ++i)
    {
        ctx->data[ctx->datalen] = data[i];
        ctx->datalen++;
        if (ctx->datalen == 64)
        {
                sha1_transform(ctx, ctx->data);
                ctx->bitlen += 512;
                ctx->datalen = 0;
        }
    }
}
```

void sha1_final(sha1_ctx *ctx, BYTE hash[SHA1_DIGEST_SIZE])

```
void sha1_final(sha1_ctx *ctx, BYTE hash[SHA1_DIGEST_SIZE])
{
        uint32_t i;

        i = ctx->datalen;

        // Pad whatever data is left in the buffer.
```

```
        if (ctx->datalen < 56)
        {
                ctx->data[i++] = 0x80;
                while (i < 56)
                        ctx->data[i++] = 0x00;
        }
        else
        {
                ctx->data[i++] = 0x80;
                while (i < 64)
                        ctx->data[i++] = 0x00;
                sha1_transform(ctx, ctx->data);
                memset(ctx->data, 0, 56);
        }

        // Append to the padding the total message's length in bits and transform.
        ctx->bitlen += ctx->datalen * 8;
        ctx->data[63] = ctx->bitlen;
        ctx->data[62] = ctx->bitlen >> 8;
        ctx->data[61] = ctx->bitlen >> 16;
        ctx->data[60] = ctx->bitlen >> 24;
        ctx->data[59] = ctx->bitlen >> 32;
        ctx->data[58] = ctx->bitlen >> 40;
        ctx->data[57] = ctx->bitlen >> 48;
        ctx->data[56] = ctx->bitlen >> 56;
        sha1_transform(ctx, ctx->data);

        // Since this implementation uses little endian byte ordering and MD uses big endian,
        // reverse all the bytes when copying the final state to the output hash.
        for (i = 0; i < 4; ++i)
        {
                hash[i] = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 4] = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 8] = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
                hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
        }
}
```

void *sha1(const void *m, size_t len, void *md, size_t md_len)

```
void *sha1(const void *m, size_t len, void *md, size_t md_len)
{
    UNUSED(md_len);
    BYTE hash[SHA1_DIGEST_SIZE];
    sha1_ctx ctx;

    sha1_init(&ctx);
    sha1_update(&ctx, (BYTE *)m, len);
    sha1_final(&ctx, hash);
    memcpy(md, hash, SHA1_DIGEST_SIZE);
    return md;
}
```

D SHA-2 Implementation Details

sha2.c

For this file, we first include the sha2.h library, then implement the various functions defined in sha2.h. To do this, we have:

// FUNCTION SHFR (SHIFT RIGHT), ROTR (ROTATE RIGHT), ROTL (ROTATE LEFT), CH (CHOICE), MAJ (MAJORITY)

#define SHFR(x, n) (x ≫ n): performs a right shift of n bits on the value x.

#define ROTR(x, n) ((x ≫ n) | (x ≪ ((sizeof(x) ≪ 3) - n))): Performs a right rotation by calculating the value x of n bits combined with another left shift calculation on x of (sizeof(x) ≪ 3) - n bits, resulting in a right rotation.

#define ROTL(x, n) ((x ≪ n) | (x ≫ ((sizeof(x) ≪ 3) - n))): a combination of two calculations: one for the left shift calculation on x (x ≪ n) and one for the right shift calculation on x (x ≫ ((sizeof(x) ≪ 3) - n), resulting in a left rotation effect.

#define CH(x, y, z) ((x & y) ∧ (∼x & z)): a macro whose operation is performed by using a XOR (∧) between (x & y) and (∼x & z), and the combination of the two resulting in (x & y) ∧ (∼x & z).

#define MAJ(x, y, z) ((x & y) ∧ (x & z) ∧ (y & z)): a macro for implementing a function that returns the majority bit.

In terms of security, these different operations allow more efficient bit manipulation.

```
// SHA256 HASH FUNCTION (CALLED BY SHA256) A VARIANT OF SHA2
#define SHA256_F1(x) (ROTR(x,  2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define SHA256_F2(x) (ROTR(x,  6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define SHA256_F3(x) (ROTR(x,  7) ^ ROTR(x, 18) ^ SHFR(x,  3))
#define SHA256_F4(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ SHFR(x,10))
```

The above code, which constitutes the essential transformation for calculating intermediate values in the SHA-256 hashing process, defines four macros SHA256_F1(x), SHA256_F2(x), SHA256_F3(x), and SHA256_F4(x) for hash functions used in the SHA-256 algorithm. Each macro performs right rotations (ROTR) and logical right shifts (SHFR) on a 32-bit integer.

- The macro SHA256_F1(x) combines three right rotations of 2, 13, and 22 bits and then applies a XOR operation on the results.
- Next, SHA256_F2(x) does the same with right rotations of 6, 11, and 25 bits.
- Then, the macro SHA256_F3(x) combines right rotations of 7 and 18 bits with a logical right shift of 3 bits.
- Finally, the macro SHA256_F4(x) combines right rotations of 17 and 19 bits with a logical right shift of 10 bits.

```
// SHA512 HASH FUNCTION (CALLED BY SHA512) A VARIANT OF SHA2
#define SHA512_F1(x) (ROTR(x, 28) ^ ROTR(x, 34) ^ ROTR(x, 39))
#define SHA512_F2(x) (ROTR(x, 14) ^ ROTR(x, 18) ^ ROTR(x, 41))
#define SHA512_F3(x) (ROTR(x,  1) ^ ROTR(x,  8) ^ SHFR(x,  7))
#define SHA512_F4(x) (ROTR(x, 19) ^ ROTR(x, 61) ^ SHFR(x,  6))
```

In this code, we have the definition of 04 macros SHA512_F1(x), SHA512_F2(x), SHA512_F3(x), and SHA512_F4(x) for hash functions used in the SHA-512 algorithm, and some of them perform right rotations (ROTR) and logical right shifts (SHFR) on a 64-bit integer.

- The macro SHA512_F1(x) combines three right rotations of 28, 34, and 39 bits and then applies a XOR (∧) operation.
- Next, the macro SHA512_F2(x) performs the same calculations with right rotations of 14, 18, and 41 bits.
- Then, the macro SHA512_F3(x) combines right rotations of 1 and 8 bits with a 7-bit shift.
- Finally, SHA512_F4(x), which is a macro performing a combination of right rotations of 19 and 61 bits with a 6-bit right shift.

```
// SHA-256 Hash Function (called by sha256) a variant of SHA2
#define UNPACK32(x, str)                        \
{                                               \
    *((str) + 3) = (uint8) ((x)      );         \
    *((str) + 2) = (uint8) ((x) >>  8);         \
    *((str) + 1) = (uint8) ((x) >> 16);         \
    *((str) + 0) = (uint8) ((x) >> 24);         \
```

UNPACK32 is a macro used in the SHA-256 algorithm to break down a 32-bit integer into four bytes. It takes an integer x and str as locations and stores the four bytes of x in the four memory locations pointed to by str. Thus, the least significant byte of x is stored at address str + 3, and the most significant byte is stored at address str + 0.

```
// SHA-256 Hash Function (called by sha256) a variant of SHA2
#define PACK32(str, x)                              \
{                                                   \
    *(x) =   ((uint32) *((str) + 3)      )          \
           | ((uint32) *((str) + 2) <<  8)          \
               | ((uint32) *((str) + 1) << 16)      \
               | ((uint32) *((str) + 0) << 24);     \
}
```

PACK32 is a macro defined in SHA-256 for converting a sequence of 4 bytes into 32 bits. It takes two pointers as arguments: str and a pointer to an integer x. It reads the four bytes pointed to by str and combines them to form a 32-bit integer. Thus, the byte at address str + 3 is the least significant byte of x, and the byte at address str + 0 becomes the most significant.

```
// SHA-256 Hash Function (called by sha256) a variant of SHA2
#define UNPACK64(x, str)                                \
{                                                       \
               *((str) + 7) = (uint8) ((x)      );      \
               *((str) + 6) = (uint8) ((x) >>  8);      \
               *((str) + 5) = (uint8) ((x) >> 16);      \
               *((str) + 4) = (uint8) ((x) >> 24);      \
               *((str) + 3) = (uint8) ((x) >> 32);      \
               *((str) + 2) = (uint8) ((x) >> 40);      \
               *((str) + 1) = (uint8) ((x) >> 48);      \
               *((str) + 0) = (uint8) ((x) >> 56);      \
}
```

This UNPACK64 macro function breaks down a 64-bit integer (x) into an 8-byte array (str), storing each byte of the integer in reverse order.

```
// SHA-256 Hash Function (called by sha256) a variant of SHA2
#define PACK64(str, x)                                      \
{                                                           \
               *(x) =   ((uint64) *((str) + 7)      )       \
               | ((uint64) *((str) + 6) <<  8)              \
           | ((uint64) *((str) + 5) << 16)                  \
               | ((uint64) *((str) + 4) << 24)              \
               | ((uint64) *((str) + 3) << 32)              \
               | ((uint64) *((str) + 2) << 40)              \
               | ((uint64) *((str) + 1) << 48)              \
               | ((uint64) *((str) + 0) << 56);             \
}
```

```
// SHA-256 Constants Table SCR
#define SHA256\_SCR(i)                          \
{                                               \
    w[i] =  SHA256\_F4(w[i -  2]) + w[i -  7]    \
        + SHA256\_F3(w[i - 15]) + w[i - 16];     \
}

// SHA-512 Constants Table SCR (which means "Schedule")
#define SHA512_SCR(i)                           \
```

```
{                                                \
    w[i] =  SHA512_F4(w[i -  2]) + w[i -  7]  \
        + SHA512_F3(w[i - 15]) + w[i - 16];   \
}
```

This SHA256_SCR macro operation performs work value calculations w in the SHA-256 algorithm. The macro takes the index (i) as an argument and updates the element w[i] using a combination of previous values of w and the SHA256_F3 and SHA256_F4 hash functions.

```
//  SHA-256 Expansion Constants Table
#define SHA256_EXP(a, b, c, d, e, f, g, h, j)               \
{                                                           \
    t1 = wv[h] + SHA256_F2(wv[e]) + CH(wv[e], wv[f], wv[g])  \
        + sha256_k[j] + w[j];                               \
    t2 = SHA256_F1(wv[a]) + MAJ(wv[a], wv[b], wv[c]);       \
    wv[d] += t1;                                            \
    wv[h] = t1 + t2;                                        \
}
```

```
// SHA-512 Expansion Constants Table
#define SHA512_EXP(a, b, c, d, e, f, g, h, j)                \
{                                                            \
    t1 = wv[h] + SHA512_F2(wv[e]) + CH(wv[e], wv[f], wv[g])   \
        + sha512_k[j] + w[j];                                \
    t2 = SHA512_F1(wv[a]) + MAJ(wv[a], wv[b], wv[c]);        \
    wv[d] += t1;                                             \
    wv[h] = t1 + t2;                                         \
}
```

Next, we initialize the constant tables for the SHA-256, SHA-384, and SHA-512 algorithms.

- Tables sha224_h0 and sha256_h0: These tables contain the initial values of the hash registers for the SHA-224 and SHA-256 algorithms, respectively. These values are used to initialize the registers before starting the hashing process.
- Tables sha384_h0 and sha512_h0: Similarly, these tables contain the initial values of the hash registers for the SHA-384 and SHA-512 algorithms, respectively. They are used to initialize the registers before starting the hashing process.
- Tables sha256_k and sha512_k: These tables contain specific constants used in the compression stages of the SHA-256 and SHA-512 algorithms. Each constant is used in one of the 64 (for SHA-256) or 80 (for SHA-512) iterations of the hashing process to securely mix the message bits.

sha256_transfert(sha256_ctx *ctx, const uint8 *message, uint64 block_nb)

sha256_transfert is a transformation function that transfers the message data into the hashing context. The steps are as follows:

- **Initialization of variables**: The function begins by declaring 32-bit arrays w and wv, as well as variables t1, t2, and a sub_block pointer. The variable i is used for loops.
- **Block processing**: The function processes each 512-bit (64-byte) block of the message. For each block, it initializes sub_block to point to the start of the current block in the message.
- **Block decomposition**: The first 16 words of 32 bits in the w array are filled using the PACK32 macro, which converts block bytes into 32-bit words.
- **Message expansion**: The remaining 48 words in w are calculated using the SHA256_SCR macro, which applies specific transformations to extend the initial 16 words to 64 words.
- **Register initialization**: The working registers wv are initialized with the current values of the hash registers ctx->h.
- **Compression**: The function performs 64 compression rounds using the SHA256_EXP macro, which applies the specific compression transformations of SHA-256. Each round uses the words in w and the constants of the algorithm to mix the bits of the working registers.

- **Update of hash registers**: After the 64 compression rounds, the hash registers ctx->h are updated by adding the values of the working registers wv.

sha512_transf(sha512_ctx *ctx, const uint8 *message, uint64 block_nb)

The sha512_transf function takes as arguments a pointer *ctx, a constant message, and block_nb for the number of blocks. It performs the necessary transformations to mix the message bits, thereby producing the SHA-512 hash of the complete message. This function is called for each 1024-bit block of the message. The structure is as follows:

- **Initialization of variables**: Arrays w and wv of 64 bits are declared, along with variables t1, t2, and a sub_block pointer. The variable i is used for loops.
- **Block processing**: Each 1024-bit (128-byte) block of the message is processed. For each block, sub_block is initialized to point to the start of the current block in the message.
- **Block decomposition**: The first 16 words of 64 bits in the w array are filled using the PACK64 macro, which converts block bytes into 64-bit words.
- **Message expansion**: The remaining 64 words in w are calculated using the SHA512_SCR macro, which applies specific transformations to extend the initial 16 words to 80 words.
- **Register initialization**: The working registers wv are initialized with the current values of the hash registers ctx->h.
- **Compression**: The function performs 80 compression rounds using the SHA512_EXP macro, which applies the specific compression transformations of SHA-512. Each round uses the words in w and the constants of the algorithm to mix the bits of the working registers. The compression rounds are then performed in a do-while loop that iterates 80 times.
- **Update of hash registers**: After the 80 compression rounds, the hash registers ctx->h are updated by adding the values of the working registers wv.

sha224(const uint8 *message, uint64 len, uint8 *digest)

sha224 is a function that encapsulates the necessary steps to compute the SHA-224 hash of a given message by initializing the context, updating the context with the message, and finalizing the calculation to obtain the digest.

sha224_init(sha224_ctx *ctx)

The calculation involves initializing the sha224_ctx context for computing the SHA-224 hash. The sha224_init function copies the initial values of the hash registers from the sha224_h0 array into the ctx context. Then, it initializes the lengths len and tot_len to zero to prepare the context for processing a new message.

sha224_update(sha224_ctx *ctx, const uint8 *message, uint64 len)

The sha224_update function, which is an update function, copies the data into the block and processes complete 512-bit blocks by calling the sha256_transf function. Thus, if the message is longer than a block, it processes the complete blocks and keeps the remaining data for the next call.

sha224_final(sha224_ctx *ctx, uint8 *digest)

The sha224_final function consists of two arguments: a *ctx pointer and a *digest pointer. It finalizes the SHA-224 hash calculation by adding padding to the data block, including a 0x80 bit and the total length of the message in bits. To complete the hash calculation, it calls the sha256_transf function and processes the padding blocks with it.

void sha256(const uint8 *message, uint64 len, uint8 *digest)

This function computes the hash of a given message. First, it initializes the context with sha256_init(&ctx), then updates the context with the message using the sha256_update(&ctx, message, len) function, and finalizes the calculation to produce the digest (an array that receives the results) with sha256_final(&ctx, digest);

sha256_init, sha384_init, and sha512_init

The three functions sha256_init, sha384_init, and sha512_init perform the same calculations as sha224_init.

sha256_update, sha384_update, and sha512_update

Similarly, the three functions sha256_update, sha384_update, and sha512_update perform the same calculations as sha224_update, with the only difference being the manipulation of the number of bytes.

sha256_final, sha384_final, and sha512_final

The same applies to the three functions sha256_final, sha384_final, and sha512_final. They perform the same calculations as sha224_final, with the only difference being the manipulation of the number of bytes and also in these calculations: block_nb = (1 + ((SHA256_BLOCK_SIZE - 9) < (ctx→len % SHA256_BLOCK_SIZE))) for SHA256, block_nb = (1 + ((SHA384_BLOCK_SIZE - 17) < (ctx->len % SHA384_BLOCK_SIZE))) for SHA384, and block_nb = 1 + ((SHA512_BLOCK_SIZE - 17) < (ctx->len % SHA512_BLOCK_SIZE)) for SHA512.

E SHA-3 Implementation Details

Five algorithms applied in Keccak's permutation

Algorithm 43 Theta Step

Input: State array $st[25]$
Output: Updated state array $st[25]$

for $i = 0$ to 4 **do**
    $bc[i] \leftarrow st[i] \oplus st[i+5] \oplus st[i+10] \oplus st[i+15] \oplus st[i+20]$
end for
for $i = 0$ to 4 **do**
    $t \leftarrow bc[(i+4) \mod 5] \oplus \text{ROTL}(bc[(i+1) \mod 5], 1)$
    for $j = 0$ to 20 by 5 **do**
        $st[j+i] \leftarrow st[j+i] \oplus t$
    end for
end for

Algorithm 44 Rho and Pi Steps

Input: State array $st[25]$
Output: Updated state array $st[25]$

$t \leftarrow st[1]$
for $i = 0$ to 23 **do**
    $j \leftarrow \text{keccakf\_piln}[i]$
    $bc[0] \leftarrow st[j]$
    $st[j] \leftarrow \text{ROTL}(t, \text{keccakf\_rotc}[i])$
    $t \leftarrow bc[0]$
end for

Note. These two algorithms are combined for performance optimization.

Algorithm 45 Chi Step

Input: State array $st[25]$
Output: Updated state array $st[25]$

for $j = 0$ to 20 by 5 **do**
    for $i = 0$ to 4 **do**
        $bc[i] \leftarrow st[j+i]$
    end for
    for $i = 0$ to 4 **do**
        $st[j+i] \leftarrow st[j+i] \oplus (\neg bc[(i+1) \mod 5] \wedge bc[(i+2) \mod 5])$
    end for
end for

Algorithm 46 Iota Step

Input: State array $st[25]$, Round index r
Output: Updated state array $st[25]$

$st[0] \leftarrow st[0] \oplus \text{keccakf\_rndc}[r]$

F PKCS#1 - RSA Cryptography Standard

- Header files
  - PKCS #1 Headers: Implements padding and mask generation functions such as MGF1 and OAEP/PSS padding.
  - Includes standard headers: `stdlib.h`, `stdio.h`, `time.h`.
  - Includes the GMP header: `gmp.h` for handling large numbers.
- Encryption Schemes
  - Function rsa_encrypt:
    - Implements RSA encryption using public keys.
    - Utilizes Optimal Asymmetric Encryption Padding (OAEP):
      - Pads the plaintext using MGF1 and SHA-256.
      - Encodes the padded plaintext into ciphertext using $c = m^e$ mod n
- Decryption Schemes
  - Function rsa_decrypt:
    - Uses private keys to recover plaintext.
    - Decodes ciphertext c using $m = c^d$ mod n.
    - Removes OAEP padding to retrieve the original plaintext.
- Signature Schemes
  - Function rsa_sign:
    - Signs a hashed message $H(m)$ with the private key.
    - Pads the hash using RSASSA-PSS or RSASSA-PKCS1-v1_5:
      - RSASSA-PSS uses MGF1 and random salt for added security.
    - Computes the signature $s = H(m)^d$ mod n.
- Verification Schemes
  - Function rsa_verify:
    - Verifies the signature s using the public key.
    - Decodes s using $H(m) = s^e$ mod n.
    - Confirms the hash matches the signed message.

Algorithm 47 EME-PKCS1 Encoding

Input: Message m (as bytes), message length m_len, encoded message em, encoded length em_len
Output: Encoded message em

Initialize random state
Set $em[0] \leftarrow 0x00$ and $em[1] \leftarrow 0x02$
for $i = 2$ to $em\_len - m\_len - 2$ **do**
    repeat
        Generate random non-zero byte for $em[i]$
    until $em[i] \neq 0$
end for
$em[em\_len - m\_len - 1] \leftarrow 0x00$
Copy message m to $em[em\_len - m\_len, \ldots, em\_len]$

Algorithm 48 PKCS#1 Encryption

Input: Public key pk, message m, ciphertext c
Output: Encrypted ciphertext c

Calculate message length m_len and key length k
if $m\_len > k - 11$ **then**
    Error: "Message is too long"
end if
Encode message using EME-PKCS1-Encode
Convert encoded message to integer $padded\_m$
Encrypt using RSA: $c \leftarrow padded\_m^e \mod n$

Algorithm 49 PKCS#1 Decryption

Input: Private key sk, ciphertext c, plaintext m
Output: Decrypted plaintext m

Decrypt using RSA: $padded\_m \leftarrow c^d \mod n$
Convert $padded\_m$ to bytes and check padding:
if Padding invalid or m length exceeds constraints **then**
    Error: "Invalid ciphertext"
end if
Extract plaintext from encoded message

Algorithm 50 EME-OAEP Encoding

Input: Message m, encoded message em, security level sec_level
Output: Encoded message em

Generate hash and padding based on sec_level
Generate random seed and calculate masks using MGF1
XOR seed and message with respective masks
Concatenate seed and masked message to form em

Algorithm 51 OAEP Encryption

Input: Public key pk, message m, ciphertext c, security level sec_level
Output: Encrypted ciphertext c

Encode m using EME-OAEP-Encode
Convert encoded message to integer $padded\_m$
Encrypt using RSA: $c \leftarrow padded\_m^e \mod n$

Algorithm 52 OAEP Decryption

Input: Private key sk, ciphertext c, plaintext m, security level sec_level
Output: Decrypted plaintext m

Decrypt ciphertext: $padded\_m \leftarrow c^d \mod n$
Convert $padded\_m$ to bytes and verify OAEP padding
if Padding invalid **then**
    Error: "Invalid padding"
end if
Extract plaintext from encoded message

Algorithm 53 EMSA-PKCS1 Encoding for Signing

Input: Message m, encoded message em, security level sec_level
Output: Encoded message em

1: Hash m using the hash function defined by sec_level
2: Pad the hash with PKCS#1 padding to fit em_len

Algorithm 54 PKCS#1 Signing

Input: Private key sk, message m, signature s, security level sec_level
Output: Signature s

1: Encode m using EMSA-PKCS1-Encode
2: Convert encoded message to integer $padded\_m$
3: Sign: $s \leftarrow padded\_m^d \mod n$

Algorithm 55 PKCS#1 Verification

Input: Public key pk, message m, signature s, security level sec_level
Output: Verification result

1: Verify: $padded\_m \leftarrow s^e \mod n$
2: Check if $padded\_m$ matches encoding of m using EMSA-PKCS1-Encode
3: **if** Match **then**
4: **return** 1 ▷ Valid signature
5: **else**
6: **return** 0 ▷ Invalid signature
7: **end if**

G MGF - Mask Generator Function

We implemented the MGF1 function for usage in RSAOAEP encryption variant and RSAPSS digital signature scheme.

```
void mgf1(unsigned char *mask, size_t mask_len, const unsigned char *seed, size_t seed_len, hash_func_t
↪  hash_function, sec_level_t sec_level);
```

While the implementation of the selected algorithm allows the usage of any hash functions, i.e. MD5, SHA-1, SHA-2, SHA-3, we found a test vector with SHA-1 to verify the correctness of our implementation.

Algorithm 56 Mask Generation Function (MGF1)

Input: Seed seed, Seed length seed_len, Mask length mask_len, Hash function H, Security level sec_level
Output: Generated mask mask of length mask_len

Allocate buffer buf of size seed_len + 4
Copy seed into buf
Select hash function H based on hash_function and security level:
if $H = \text{MD5}$ **then**
    hash_len $\leftarrow 16$
else if $H = \text{SHA-1}$ **then**
    hash_len $\leftarrow 20$
else if $H = \text{SHA-2}$ **then**
    Select hash_len based on sec_level
else if $H = \text{SHA-3}$ **then**
    Select hash_len based on sec_level
else
    Exit with error: Invalid hash function
end if
if mask_len $> 2^{32} \times$ hash_len **then**
    Exit with error: Mask too long
end if
Compute number of blocks: num_blocks $\leftarrow \lceil$mask_len/hash_len$\rceil$
Allocate buffer md of size hash_len
for $i = 0$ to num_blocks $- 1$ **do**
    Append counter i to buf
    Compute hash: H(buf)
    Copy hashed output into mask, handling length constraints
end for
Free allocated memory
return mask

H PKCS#7 - Cryptographic Message Syntax Standard

- Definition
  - PKCS #7 Headers: Implements input padding for block ciphers.
  - Defined in `conversion.h`.
- Usage
  - Used in: DES, AES, CBC-MAC. It used for ensure correct block size input (and unpadding output) for these algorithm.

Algorithm 57 PKCS7 Padding

Input: *input*, *len*, *block_size*
Output: *output*, *out_len*

1: $out\_len \leftarrow len + (block\_size - len \mod block\_size)$
2: $padding\_value \leftarrow block\_size - (len \mod block\_size)$
3: Allocate memory for *output* of size $out\_len + 1$
4: Copy *input* to *output* up to index *len*
5: **for** $i = len$ to $out\_len - 1$ **do**
6: $output[i] \leftarrow padding\_value$
7: **end for**
8: $output[out\_len] \leftarrow' \backslash 0'$
9: **return** *output*

Algorithm 58 PKCS7 Unpadding

Input: *input*, *len*, *block_size*
Output: *output*, *out_len*

1: **if** $len \mod block\_size \neq 0$ **then**
2: **Exit with error:** Invalid input length
3: **end if**
4: $padding\_value \leftarrow input[len - 1]$
5: **if** $padding\_value > block\_size$ **then**
6: **Exit with error:** Invalid padding value
7: **end if**
8: $out\_len \leftarrow len - padding\_value$
9: **for** $i = out\_len$ to $len - 1$ **do**
10: **if** $input[i] \neq padding\_value$ **then**
11: **Exit with error:** Invalid padding value
12: **end if**
13: **end for**
14: Allocate memory for *output* of size $out\_len + 1$
15: Copy *input* to *output* up to index out_len
16: $output[out\_len] \leftarrow' \backslash 0'$
17: **return** *output*

I AES Constants

- **S-Box:**

```
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
10 ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
20 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
30 04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
40 09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
50 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
60 d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
70 51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
80 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
90 60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
a0 e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
b0 e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
c0 ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
d0 70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e0 e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
f0 8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```

- **Inverse S-Box:**

```
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00 52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
10 7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
20 54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
30 08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
40 72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
50 6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
60 90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
70 d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
80 3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
90 96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
a0 47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
b0 fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
c0 1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
d0 60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
e0 a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
f0 17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

- **Circulant MDS matrix:**

```
2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
```

- **Inverse Circulant MDS matrix:**

```
14 11 13 9
9 14 11 13
13 9 14 11
11 13 9 14
```

J RFC3526 - MODP Diffie-Hellman groups for IKE

To match our definition of security level across different cryptography functionalities in our library, we utilized two modular exponential groups proposed in [KK03]:

- **2048-bit MODP Group**

  This group is assigned id 14.

  The prime is:

  $$2^{2048} - 2^{1984} - 1 + 2^{64} \times \{2^{1918}\pi\} + 124476$$

  Its hexadecimal value is:

  ```
  FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
  29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
  EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
  E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CBC F406B7ED
  E3E386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
  C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
  83655D23 DCA3AD96 16C2F356 208552B8 9ED52907 7096966D
  670C354E A4BC9804 F1746C08 CA18217C 32905E46 2E36CE3B
  E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
  DE2BCBF6 95581713 3995497C EA956AE5 15D22618 98FA0510
  15728E5A 8AACAA68 FFFFFFFF FFFFFFFF
  ```

  The generator is: 2.

- **3072-bit MODP Group**

  This group is assigned id 15.

  The prime is:

  $$2^{3072} - 2^{3008} - 1 + 2^{64} \times \{2^{2942}\pi\} + 1690314$$

  Its hexadecimal value is:

  ```
  FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
  29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
  EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
  E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CBC F406B7ED
  E3E386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
  C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
  83655D23 DCA3AD96 16C2F356 208552B8 9ED52907 7096966D
  670C354E A4BC9804 F1746C08 CA18217C 32905E46 2E36CE3B
  E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
  DE2BCBF6 95581713 3995497C EA956AE5 15D22618 98FA0510
  15728E5A 8AACAA68 AD33170D 04507A33 A85521AB DF1CBA64
  ECFB8504 58D08B93 A8EA7157 5D06C7D B3970F85 A6E14C7
  ABF5AE8C D80933D7 18EC94E0 4A25619D CEE3D226 1AD2E6B
  F12FFA0D D98A0864 D8760273 3EC86A64 521F2B18 177B200C
  BBE11757 7A61D56C 779838C0 BDA946E2 082E47AE 7045AB31
  3B4DB5BF E0F01D8E 4B8212D0 A93AD2CA FFFFFFFF FFFFFFFF
  ```

  The generator is: 2.

K Elliptic Curve Explicit-Formulas Database

We implemented the following explicit-formulas for the addition and doubling algorithms of the elliptic curves. All of these formulas are specified in the database [BL].

Weierstrass Curve

Addition with $a = -3$

- Addition Formulas: `add-2015-rcb`.
- Assumptions: $b3 = 3b$.
- Cost: $12M + 2 * b3 + 3 * a + 23\text{add}$

Addition with $a \neq -3$

- Addition Formulas: `madd-2015-rcb`.
- Assumptions: $Z2 = 1 \quad \& \quad b3 = 3b$.
- Cost: $11M + 2 * b3 + 3 * a + 17\text{add}$

Doubling

- Addition Formulas: `dbl-2015-rcb`.
- Assumptions: $b3 = 3b$.
- Cost: $8M + 3S + 2 * b3 + 3 * a + 15\text{add}$

Montgomery Curve

Differiential Addition

- Addition Formulas: `dadd-1987-m-3`.
- Cost: $4M + 2S + 6\text{add}$

Doubling

- Addition Formulas: `dbl-1987-m-3`.
- Assumptions: $4 * a24 = a + 2$.
- Cost: $2M + 2S + 1 * a24 + 4\text{add}$

Twisted Edwards Curve

Addition

- Addition Formulas: `add-2008-bbjlp`.
- Cost: $10M + 1S + 1 * a + 1 * d + 7\text{add}$

Doubling

- Addition Formulas: `dbl-2008-bbjlp`.
- Cost: $3M + 4S + 1 * a + 6\text{add} + 1 * 2$