Goal

In this Lab, you will write a program that produces and verifies RSA and DSA signatures.

0. Before starting

To be sure to use the latest version, download Lib.zip, extract the files and build the library.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/Lib.zip
```

```
$ unzip Lib.zip; cd Lib; make clean; make; cd ..
```

Download the source files Lab7.zip, unzip them and move to the corresponding directory.

```
$ wget https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_7/Lab7.zip
```

```
$ unzip Lab7.zip; cd Lab7
```

As you see, the folder contains programs (`.c` and `.h` files) but also data files contained in the `data` folder.

You are now ready to code. Open your favorite source-code editor. If you are used to working with VS Code, this can be done with:

```
$ code .
```

In this lab, make sure to **respect the output format very carefully**.

I. Signing with RSA

1. RSA public key files

RSA public keys are stored using the following format, with the integers `N` and `e` represented as hexadecimal strings with a leading `0x`:

```
$ cat data/RSA_pub_ref.txt
```

```
#RSA Public key (256 bits):
N = 0xd5d3ffc9cbed4fe82f31e7eb0c5fd9240f5602e471c6aaba51c9b226b4675eeb
e = 0x6057fcff1d4a2f3bc7f562e82026a08155a255133b79ec37c8df421b5c7938a1
```

RSA private keys are stored using the following format, with the integers `N` and `d` represented as hexadecimal strings with a leading `0x`:

```
$ cat data/RSA_sec_ref.txt
```

```
#RSA Secret key (256 bits):
N = 0xd5d3ffc9cbed4fe82f31e7eb0c5fd9240f5602e471c6aaba51c9b226b4675eeb
d = 0x71306deaf57708eb08ae09a8eb3c1c680b1b249e1c670f986cd55f862c3b4635
```

In the file `sign.c` complete the function which creates two files, one for the public key and one for the secret key.

```
int RSA_generate_key_files(
      const char *pk_file_name,
      const char *sk_file_name,
      size_t nbits,
      int sec,
      gmp_randstate_t state)
```

Hint: to write Hexadecimal strings in a file, use

```
gmp_fprintf(file, "... %#Zx ...", N);
```

You can test your function using:

```
$ make
```

```
$ ./test_lab7 1
```

This should create two files `RSA_pub.txt` and `RSA_sec.txt`. These are your keys! And the expected output is:

```
------------------ Test 1 -----------------

Key generation...
Done

Key generation [OK].
```

Upload your file `sign.c` on VPL.

2. Parsing key files

(nothing to program)

We provide you with the following function which fills in `N` and `ed` from the file `key_file_name`.

```
void RSA_key_import(mpz_t N, mpz_t ed, const char *key_file_name)
```

The variable `ed` refers to `e` if we parse a public key and `d` if we parse a secret key.

Hint: to parse hexadecimal strings and fill in an `mpz_t a`, use

```
gmp_fscanf(file, "%Zx", a);
```

You can test your function using:

```
$ make
```

```
$ ./test_lab7 2
```

3. Signing and verifying a buffer

Complete the following functions.

```
int RSA_sign_buffer(mpz_t sgn, buffer_t *msg, mpz_t N, mpz_t d)
```

```
int RSA_verify_signature(mpz_t sgn, buffer_t *msg, mpz_t N, mpz_t e)
```

The first one fills in `sgn` from the buffer `msg` and the secret key `(N, d)`. The second one verifies this signature using the public key `(N, e)` and returns `1` if verification succeeds and `0` if not.

Hint 1. You will need a hash function. We give you an implementation of SHA3. You can hash as follows:

```
void buffer_hash(buffer_t *hash, int length, buffer_t *msg)
```

For the output length, you can take, the byte size of `N` (minus 1). This can be computed using function `hash_length(mpz_t N)` given in `sign.c`

Hint 2. To convert a `buffer` into an `mpz_t` you can use the function

```
mpz_import(output, length, 1, 1, 1, 0, buffer_input.tab);
```

See the documentation of import functions of gmp for further details.

You can test your function using:

```
$ make
```

```
$ ./test_lab7 3
```

You should get

```
------------------ Test 3 -----------------

Message : Fear is the path to the dark side. Fear leads to anger. Anger leads to hate. Hate leads to suffering.

Signature : 0x149a2fc00ab718312145b27a97df2945ed419b4ebeec28ff14e863b01cb9ffd9

Verification:   [OK]
```

Upload your file `sign.c` on VPL.

4. Signing and verifying a file

(nothing to program)

Using the functions given in your files, it is now possible to sign files. For instance a recipe of duck salad (see the file in `data` folder) can be signed by doing:

```
$ ./test_lab7 4
```

This should output:

```
------------------ Test 4 -----------------

Message :

1. Heat oven to 200C/fan 180C/gas 6.
Score the skin of the duck breasts and season.
Heat a non-stick frying pan over a high heat, add the duck, skin-side down,
and cook for 4 mins or until the skin is crisp.
Turn over and quickly brown the underside, then transfer to a baking tray.

2. Mix the dressing ingredients together and spoon all but 2 tbsp of it over the duck.
Roast the duck for 10 mins for pink, longer if you prefer.
Remove from the oven and allow to rest for 4 mins, then slice into strips.

3. Toss together the salad, tomatoes, spring onions and duck slices.
Drizzle over the remaining dressing and serve.
-------------------------------------------------



-----------------------------------------------

#RSA signature
S = 0x1778a32babfcd76afbf3dec0bfd1625c550b98789b3a872dcaac648e18c9ac97

-----------------------------------------------

Signature verification...

[OK]
```

Your signature of the file is in `Duck_salad_RSA_signature.txt` You can compare it with a reference file.

```
$ diff Duck_salad_RSA_signature.txt data/Duck_salad_RSA_signature_ref.txt
```

Nothing should be printed in the console (i.e. no diff).

Upload your file `sign.c` on VPL.

II. DSA signatures

5. Prime numbers generation

First to generate DSA signatures, we need a good pair of primes `(p, q)`. Such that `q` divides `p-1`. To do this, first generate `q`, then generate `p` as a multiple of `q` plus one and check whether `p` is prime. If it fails try again until it succeeds.

In the file `dsa.c`, complete the function `generate_pq`. It fills in `p, q` when `psize, qsize` are given.

You can test your function using:

```
$ make
```

```
$ ./test_lab7 5
```

Upload your file `dsa.c` on VPL.

6. Full key generation

Complete the function `int dsa_generate_keys` which performs the key generation. That is, fills in `p, q, a, x, y` when `psize, qsize` are given.

Hint: to select *a*, find a random *h* such that *a = h^((p-1)/q) mod p* is not 1; *a* will be a primitive *q*-th root of 1.

You can test your function using:

```
$ make
```

```
$ ./test_lab7 6
```

Upload your file `dsa.c` on VPL.

7. Sign a string and verify

Complete the functions `int dsa_sign_buffer` and `int dsa_verify_buffer`.

- The first one fills in the fields `r, s`, which sign the buffer `msg` with the secret key `p, q, a, y`.
- The second one verifies the signature and returns `1` if it succeds, `0` otherwise.

Hint. You need to hash the message in order to get a big integer of size less than `q`.

You can test your function using:

```
$ make
```

```
$ ./test_lab7 7
```

Upload your file `dsa.c` on VPL.

8. With files

(nothing to program)

If your primitive works, you can generate key files using

```
$ ./test_lab7 8
```

and then sign a file using

```
$ ./test_lab7 9
```

9. Attack on DSA

As explained in the slides of today's lecture, if the random term `k` in the creation of the signature is used to sign two distinct messages, then it is possible to recover the secret key `x`.

First, in the file `attack_dsa.c`, complete the following function which solves the linear system as in the slides.

```
solve_system_modq(mpz_t x, mpz_t r1, mpz_t s1, mpz_t r2, mpz_t s2, mpz_t h1, mpz_t h2, mpz_t q)
```

You can test your function using:

```
$ make
```

```
$ ./test_lab7 10
```

It should output

```
------------------ Test 10 -----------------

Test solving system...
Linear system :
0 k - 3 x = 7
8 k - 6 x = 2

Candidate for secret key obtained from the attack:
x = 5
```

```
Linear system :
1 k - 3 x = 7
3 k - 1 x = 2

Candidate for secret key obtained from the attack:
x = 10

[OK]
```

Then, complete the function `dsa_sign_dummy` which makes the same job as `dsa_sign_buffer` but the random generation of `k` is replaced by a prescribed `k` as an argument (and `state` is no longer an argument of the function).

Finally complete the function `dsa_attack` and test it with

```
$ ./test_lab7 11
```

Upload your file `dsa.c` on VPL.