Exercise Sheet Signatures

Solve the following exercises and submit them until the communicated date.

LB-S 00. (not to be submitted)

(*Code::Blocks* template *libsodium project*)

a) Create a new project based on the template project for *libsodium* and execute it with your first name as an argument. The output of the program is the SHA-256 hash of the passed argument. Verify the output with the command `echo -n <first name> | sha256sum` (without angle brackets!) on the command line.

b) Adapt the code from a) so that SHA-512 is used instead of SHA-256. Using the documentation of *libsodium*, go through the code step by step and adapt the program accordingly. For verification, use `sha512sum` instead of `sha256sum` on the command line – the remaining usage is unchanged.

Hint: You can find the documentation of libsodium *at* `https://download.libsodium.org/doc/advanced/sha-2_hash_function.html`

LB-S 01. (*Code::Blocks* template *GMP and libsodium project*)

Using the *GMP* and *libsodium*, write a program which implements the following functions:

- `Sign` to sign a message `message` with the secret key `(d, N)` from an RSA key pair. The signature is output in decimal representation to `std::cout` in **exactly** the following format (example output) **without** additional output:

  ```
  4272599298832472[...]
  ```

- `Verify` to verify the validity of a signature `signature` corresponding to the message `message` and the public key `(e, N)` from an RSA key pair. The validity is indicated through the text output *Signature valid.* or *Signature invalid.* to `std::cout`, respectively.

The program is supposed to be called with four or five parameters via the command line as follows: `./test Sign <message> <d> <N>` or `./test Verify <message> <signature> <e> <N>`, respectively. For the creation of this program, combine your code from exercise 00. b) for the computation of the hash and your code from LB-PKC 01. for the encryption or decryption with RSA, respectively. Test your program with an RSA key pair, e.g., created with your program from LB-PKC 02. with a key length of 2,048 bits.
Hint: In order to convert the hash determined with libsodium *into a representation which is compatible for computations with the* GMP*, the array returned*

from the hash function must be represented as a number byte by byte and then be interpreted in its entirety as a (large) number. You can use the following function a variation thereof in order to achieve this:

```
1  template <size_t N> //Number of bytes (array size)
2  void libsodium_to_GMP(const unsigned char (&libsodium_value)
       ↪ [N], mpz_class &GMP_value)
3  {
4    GMP_value = 0;
5    for (const auto &libsodium_byte : libsodium_value)
6    {
7      GMP_value *= 256;
8      GMP_value += libsodium_byte;
9    }
10 }
```

Example invocation:

```
1  unsigned char hash[crypto_hash_sha512_BYTES];
2  /* TODO: Calculate hash as usual */
3  mpz_class hash_value;
4  libsodium_to_GMP(hash, hash_value);
```

LB-S 02. (not to be submitted)

Form groups of two as determined by the lecturers and verify the correctness and interoperability of your programs from example 01. through an e-mail exchange of signed messages. To do so, person P_A generates an RSA key pair (d_A, e_A, N_A) as well as an arbitrary message m_a, both of which are used to generate a signature s_a using the implementation of 01. (of P_A). Subsequently, P_A sends m_a, s_a and the public key (e_A, N_A) to person P_B via e-mail. This person passes the transmitted information to the implementation (of P_B) in order to verify the signature. Subsequently, exchange the roles of P_A and P_B and repeat the test.

LB-S 03. (*Code::Blocks* template *libsodium project*)

Write a program which, analogously to example 01. can sign and verify messages. Instead of your own implementations, use only the signing functionality of *libsodium*. Use the so-called *Combined Mode* in which the message to be signed it output together with the signature, and be guided by the documentation (`https://download.libsodium.org/doc/public-key_cryptography/public-key_signatures.html`) for the implementation. Note that, in this mode, the signature and the plaintext of the message are both contained in the output combined, and that only one single command line parameter is required for both them for the verification, i.e., the verification expects only three command line parameters in total.

The key generation is supposed to happen directly during signing, where the public key is output together with the signature in **exactly** the following format:

```
Signed message: d584bb849f3a8d96[...]
Public key: 8ab03757373db7a0[...]
```

Note that *libsodium* uses elliptic curves instead of RSA for both signing and verifying the signature, which is why the key lengths are shorter and the keys consist only of one single value.
Hints: For the output of the signature and the public key, use the loop for the hexadecimal output from example 00. To read in hexadecimal columns of digits in a format which is compatible with libsodium*, you can use the following function or a variation thereof:*

```
1  #include <string>
2
3  bool HexStringToArray(const std::string &hex_string,
       ↪ unsigned char array[], const size_t array_size)
4  {
5    if (hex_string.length() != 2 * array_size)
6      return false;
7    for (size_t i = 0; i < array_size; i++)
8    {
9      const std::string str_part(hex_string.c_str() + 2 * i,
           ↪ 2); //Process 2 characters (one byte) at a time
10     try
11     {
12       const auto byte = std::stoul(str_part, nullptr, 16);
13       array[i] = byte;
14     }
15     catch (...)
16     {
17       return false;
18     }
19   }
20   return true;
21 }
```