Exercises on Public Key Cryptography

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

LB-PKC 01.

The (encrypted) ciphertext c of a message m with a public key (e, N) in RSA is defined as follows: $c \equiv m^e \mod N$. Analogously, the decryption with a private key (d, N) is defined as follows: $m \equiv c^d \mod N$.
Write a program which implements encryption and decryption with RSA based on the *Cryptographic program* template in *Code::Blocks* in the *Programming VM*. The program has to expect three command line arguments in the following order: the message m to be encrypted (or decrypted) as a value, an exponent (e or d) and the modulus (N). The encrypted or decrypted value must be printed to `std::cout`. Use the `mpz_powm` function to compute powers within a modulus and test your program with the example values $m = 7, e = 29, d = 85$, $N = 391$.
Hint: Use instances of the `mpz_class` class to perform arithmetic on large numbers. Read the short documentation for this class (`https://gmplib.org/manual/C_002b_002b-Interface-General#C_002b_002b-Interface-General`) and take special note of the `mpz_class::get_mpz_t` method which is required to pass `mpz_class` instances to `mpz_powm` and similar functions. Instead of assigning character strings (e.g., command line arguments) to `mpz_class` instances like in the linked example, you may use one of the `mpz_class` constructors which take a character string as its argument (see `https://gmplib.org/manual/C_002b_002b-Interface-Integers#C_002b_002b-Interface-Integers`). Evaluating `mpz_class` instances during debugging is possible by entering `variable_name.get_str(10)` into the Watch window.

Example calls (with command line arguments) for testing:
`./test 7 29 391` must print the output `74`.
`./test 74 85 391` must print the output `7`.

LB-PKC 02. (<u>not</u> to be submitted)

Generate a personal, self-signed X.509 certificate which contains the public key of an RSA key pair. Simultaneously, the corresponding private key has to be created (with the same command, **without** line breaks):

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.crt
```

If necessary, adapt the output paths of both files (`key.pem` and `cert.crt`). Enter meaningful values when prompted for data and remember the entered password (*keyphrase*). The password must be at least four characters long.

LB-PKC 03.

Modify your program from exercise 01. so that the public key $pk := (e, N)$ and the private (secret) key $sk := (d, N)$ are not passed as command line ar-

guments, but instead read from a certificate or key file, respectively. The path to the respective file has to be provided as a command line argument, followed by a message (value) to be encrypted or decrypted, respectively. To distinguish between encryption and decryption, an additional first command line argument `Encrypt` or `Decrypt` has to be provided. Make use of the implemented helper functions available in the project to read the keys from the respective files and use the files generated in exercise 02. to test your program.
Note: The Cryptographic program *template in* Code::Blocks *contains two helper functions* `ReadPublicKeyFromFile` *and* `ReadPrivateKeyFromFile` *to read keys from a certificate or private key file, respectively (see the* `certhelp.h` *header file included in the project). The usage of these functions is self-explanatory.*

Example calls (with command line arguments) for testing:
`./test Encrypt cert.crt 12345` prints a value, referred to as *output1*.
`./test Decrypt key.pem` *`output1`* must print the output `12345`.
Note: The example calls assume a corresponding certificate (`cert.crt`*) and private key file (*`key.pem`*) in the same directory as the program. In the decryption call,* `output1` *is a place holder and* ***not*** *a verbatim argument.*

LB-PKC 04. (bonus exercise – voluntary)

Modify your program from exercise 03. so that it is encrypts and decrypts character strings instead of numerical values. In order to avoid issues with non-printable characters, the message (when encrypting) or ciphertext (when decrypting) must not be passed as a command line argument, but read from a file instead whose path is passed as a command line argument. Similarly, the ciphertext (when encrypting) or the message (when decrypting) must not be output onto the console, but written into a file instead whose path is passed as a command line argument.
For the conversion between character strings and numerical values, multiple approaches exist. One of the simplest is to treat each character as a number between 0 and 255 (ASCII characters only use the lower half of this range when assigning each character its position in the ASCII table, ciphertexts use the full range). For example, the character `A` is converted to the value 65 and vice versa. The values of multiple converted characters can be interpreted as one larger number in base 256, e.g., the character string `ABC` is converted to $65 \cdot 256^2 + 66 \cdot 256^1 + 67 \cdot 256^0 = 4,276,803$ and vice versa.
Note: For reading from and writing to files, the `ifstream` *and* `ofstream` *classes (both require the* `fstream` *header) can be used. Files with ciphertext strings must always be treated as binary files by using the corresponding stream file mode. For simplicity, all files may be treated as binary files.*

Example calls (with command line arguments) for testing:
`./test Encrypt cert.crt input.txt output.txt` creates `output.txt`.
`./test Decrypt key.pem output.txt check.txt` creates `check.txt` which is identical to `input.txt`.