Assignment 4

- You should have already installed and become familiar with the **Ubuntu Mate 20.04 Operating System(OS).** You can do this either:
  (1) with dual boot i.e. if you already have an OS such as MS Windows, MAC OS, or another linux distribution, just install Ubuntu Mate 20.04 as a secondary OS, or,
  (2) by installing first a Virtual machine (e.g. the Oracle VM VirtualBox is freely available) in your existing OS, and then within the VirtualBox installing the Ubuntu Mate 20.04, or,
  (3) by installing Ubuntu Mate 20.04 as your primary OS.
- If you prefer install the **Ubuntu 20.04 OS** instead of the Ubuntu Mate.
- You should have already installed the gcc compiler:
  ```
  sudo apt update
  sudo apt install build-essential
  ```
- Develop you C code using one of your favorite editors such as gedit, pluma or vi.

Introduction

In this assignment you are going to develop an **asymmetric encryption tool in C** from scratch. The purpose of this assignment, now that you are familiar with implementing simple ciphers as well as using real encryption toolkits, is to provide you the opportunity to get familiar with the internals of a popular encryption scheme, namely RSA. The tool will provide RSA key-pair generation, encryption and decryption.

More specifically, you are going to use the basic theory behind RSA and asymmetric encryption in order to develop the basics that compose a simple RSA toolkit, such as prime number generation and so on.

Task A

[Key Derivation Function (KDF)]

In this task you will implement an RSA key-pair generation algorithm. In order to do so, you first need to study RSA's internals: https://en.wikipedia.org/wiki/RSA_(cryptosystem) . Moreover, you need to develop a function that implements the Sieve Of Eratosthenes: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes . This material serves just as a reference point. You can also use any other source you wish. The key generation process will be the following:

1. Generate a pool of primes using the Sieve Of Eratosthenes. For your convenience, the sieve's limit is defined in the provided file, `rsa.h`.
2. Pick two random primes from the pool. Lets name them **p** and **q**.
3. Compute **n** where **n = p * q**.
4. Calculate **fi(n)** where **fi(n) = (p - 1) * (q - 1)**. This is Euler's totient function, as described in the original RSA paper "*A Method for Obtaining Digital Signatures and Public-Key*

Cryptosystems”. You may find out that other implementations use different totient functions. However, for this implementation, we are going to use Euler’s.

5. Choose a prime number **e** where **(e % fi(n) != 0) AND (gcd(e, fi(n)) == 1)** where gcd() is the Greatest Common Denominator.
6. Choose **d** where **d** is the **modular inverse of (e,fi(n))**.
7. The public key consists of **n** and **d**, in this order.
8. The private key consists of **n** and **e**, in this order.

Hints:

- Use `size_t` when declaring the variables described above
- The key file should just contain the two numbers, (e.g. **n** and **d**) so it just contains two `size_t` variables
- For the **modular inverse,** use whichever implementation you wish. It is not necessary to use the Extended Euclidean Algorithm (eea).
- To sieve should be able to operate with any limit. After you have implemented the function, use a predefined limit for your convenience; this way you will have a certain range of “small primes”. This will help you in managing them easily for the generation of key-pair. The function should return the array of primes, and the 2nd argument you can use it to return the size of the table of primes.

Task B

[Data Encryption]

Develop a function that provides RSA encryption functionality, using the keys generated in the previous step. This function reads the data of an input file and encrypts them using one of the generated keys. Then, it stores the ciphertext to an output file. For each character (1-byte) of the plaintext, the tool generates an 8-byte ciphertext (`size_t` on 64-bit machines). For example, if the plaintext is “`hello`” then the 5 bytes (5 chars) of the plaintext will produce 40-bytes (`5 * sizeof(size_t)`) of ciphertext.

Task C

[Data Decryption]

Implement a function that reads a ciphertext from an input file and performs RSA decryption using the appropriate one of the two keys, depending on which one was used for the ciphertext encryption. The keys will be generated using the KDF described in Task A. When the decryption is over, the function stores the plaintext in an appropriate output file.

IMPORTANT: To successfully decrypt the data, you have to use the appropriate key. If the ciphertext is encrypted using the public key, you have to use the private key in order to decrypt the data and vice versa. Also, every 8-bytes of the ciphertext produce a 1-byte plaintext. For example, a 40-byte ciphertext will produce a 5-byte plaintext.

Task D

[Using the tool]

Once you have implemented all the above functionality for your tool, use it to do the following operations, on the `.txt` files provided for validation and testing purposes:

1. Encrypt the file “`hpy414_encryptme_pub.txt`” using the `hpy414_public.key`. Name the output as `TUC<AM>_encrypted_pub.txt`.
2. Decrypt the file “`hpy414_decryptme_pub.txt`” using the `hpy414_public.key`. Name the output as `TUC<AM>_decrypted_pub.txt`.
3. Encrypt the file “`hpy414_encryptme_priv.txt`” using the `hpy414_private.key`. Name the output as `TUC<AM>_encrypted_priv.txt`.
4. Decrypt the file “`hpy414_decryptme_priv.txt`” using the `hpy414_private.key`. Name the output as `TUC<AM>_decrypted_priv.txt`.

Hints:

- Remember that the key files contain just two `size_t` values. The first 8 bytes represent **n** while the next 8 bytes represent **e** or **d**, depending on the key.

Tool Specifications

To assist you in the development of this tool, we provide a basic skeleton of the tool. We also provide some helper functions, used to print the plaintext as a string and the bytes of the ciphertext and keys in a human readable form. The tool will receive the required arguments from the command line upon execution as such:

Options:

`-i`	`path`	Path to input file
`-o`	`path`	Path to output file
`-k`	`path`	Path to key file
`-g`		Perform RSA key-pair generation)
`-d`		Decrypt input and store results to output
`-e`		Encrypt input and store results to output
`-h`		This help message

The arguments `"i", "o"` and `"k"` are always required when using `"d"` or `"e"`
Using `-i` and a path the user specifies the path to the input file.
Using `-o` and a path the user specifies the path to the output file.
Using `-k` and a path the user specifies the path to the key file.
Using `-g` the tool generates a public and a private key and stores them to the public.key and private.key files respectively.
Using `-d` the user specifies that the tool should read the ciphertext from the input file, decrypt it and then store the plaintext to the output file.
Using `-e` the user specifies that the tool should read the plaintext from the input file, encrypt it and store the ciphertext to the output file.

Example:
```
./assign_3 -g
```

The tool will generate a public and a private key and store them in the files `public.key` and `private.key` respectively.

Example:
```
./assign_3 -i plaintext.txt -o ciphertext.txt -k public.key -e
```

The tool will retrieve the public key from the file `public.key` and use it to encrypt the data found in "`plaintext.txt`" and then store the ciphertext to "`ciphertext.txt`"

IMPORTANT

Even if you chose to redesign your tool from scratch and not use the provided skeleton, you have to support the exact command line options described above.

Important notes

1. Fist study carefully the operation of RSA and the Sieve Of Eratosthenes.
2. You should implement RSA **from the scratch using only the standard libraries** of C.
3. You need to submit all the source code of your tool, a **"Makefile"**, all the files you used or generated in **Task D**, and a **"README.txt"** file that explains your implementation and what you didn't implement and why. Place all these files in a folder named <yourlastnameAM>_assign4, and then compress it as a .zip file that you will upload to eclass. For example: christodoulou2018123456_assign4.zip.
4. The README.txt file is important to submit.
5. Very important: execute the command `gcc --version` and write whatever the output is into your README.txt file, e.g. "gcc (Ubuntu 9.3.0-10ubuntu2~20.04)"
6. Do **not** copy-paste code from online examples, we will know ;)
7. The tool's skeleton provided with this assignment is just an example. Feel free to define your own functions or change the signatures of the given functions. You can even re-design it from scratch. However, the switches defined above should be kept.
8. The ciphertext and the keys are just bytes. This means that they do not terminate with \0. Don't try to print them as strings, use the provided functions, if needed.
9. You might need the command "`hexdump`" for getting the key from the given binary files. This, by default prints in HEX the bytes in little-endian representation.
10. Your solution should comply with the requirements of the assignment. Use wikipedia for assistance only.
11. Submitted code will be tested using plagiarism-detection software.