- First install the **Ubuntu Mate 20.04 Operating System(OS) 20.04.** 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 Ubuntu Mate.
- Then install the gcc compiler:

```
sudo apt update
sudo apt install build-essential
```

- Start developing in C code using one of your favorite editors such as gedit, pluma or vi.

Introduction

In this assignment you are going to implement from scratch, using C, a simple cryptographic library named simple_crypto. The cryptographic library will provide three basic but fundamental cryptographic algorithms, (i) **One-time pad**, (ii) **Caesar's cipher** and (iii) **Vigenère's cipher**. The main purpose of this assignment is to offer you the opportunity to get familiar with the implementation and internals of such simple ciphers and help you understand the development decisions and tricks that developers have to deal with when implementing security critical functions that, at first, seem trivial to develop.

The simple_crypto library will consist of two files. The "simple_crypto.h", which contains the C function declarations and any macro definitions you think are important and the "simple_crypto.c" file, containing the implementation of the above algorithms.

One-time pad

The One-Time-Pad (OTP) algorithm is a very simple but yet very strong algorithm in the sense that it can not be cracked even with post-quantum techniques. The algorithm pairs each plaintext with a random secret key (also referred to as a one-time pad). Specifically, each bit or byte/character of the plaintext is combined with the corresponding bit or byte/character from the random secret key. One-time pad requires that the secret key is of the same size or longer than the plaintext.

Implementation details:
In order to generate a random secret key you will use a pseudorandom generator, such as /dev/urandom. The pseudorandom generator will read N random characters from /dev/urandom, where N is the number of bytes/characters found in the plaintext. Then, the algorithm will encrypt each byte/character of the plaintext by XOR-ing it with the corresponding byte/character of the random secret key.

Since /dev/urandom will return a new random value upon each read, you will first need to generate an appropriate sized random secret key and store it in memory in order to successfully decrypt the encrypted message. For this functionality you can develop your own separate function or macro. Also, since the usage of /dev/urandom is our suggested pseudorandom generator, you are advised to use a Linux-based system for the development and testing of the OTP algorithm. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the random secret key and should return the result of the operation. Special characters, such as “!”, “@”, “*”, etc. that are not part of the english alphabet should be **skipped** as if the character set only consists of numbers 0-9 followed by uppercase characters A-Z and lowercase characters a-z. The same applies for all the rest of the printable and non-printable ASCII characters such as “\n”, “\t”, “\0” etc. Notice that XOR-ing specific characters together might result in non-printable characters or even “\0”. For this reason you should think around this problem when handling and printing any results. Also note that each time you access /dev/urandom you will receive a different key. For this reason the encrypted text deriving from the same plaintext is going to change across executions. Also, for the same reason, you should store the key used for a message encryption in order to successfully decrypt it.

Caesar's cipher

This technique is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each byte/character of the plaintext is replaced by a byte/character found at some fixed number of positions down the alphabet/ASCII set. For example, given the ASCII character set, a shift of 3 will replace the letter “A” of the plaintext with the letter “D” at the ciphertext. Also, a shift of 4 will encrypt the plaintext “hello” as “lipps”. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the random secret key and should return the result of the operation.

Implementation details:
The implementation should support numbers, uppercase and lowercase characters. Special characters, such as “!”, “@”, “*”, etc. that are not part of the english alphabet should be skipped as if the character set only consists of numbers 0-9 followed by uppercase characters A-Z and lowercase characters a-z. The same applies for all the rest of the printable and non-printable ASCII characters such as “\n”, “\t”, “\0” etc. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as a positive number indicating the number of shifted positions and should return the result of the operation.

Vigenère's cipher

The Vigenère's cipher encrypts an alphabetic plaintext using a series of interwoven Caesar's ciphers. In order to encrypt a message, the algorithm uses a table of alphabets, namely tabula recta, which has the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. A visual representation of this table is the following:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

The alphabet used at each point depends on a keyword, repeated in order to make a key that matches the size of the plain-text. For example, if the plaintext is "ATTACKATDAWN" and the keyword is "LEMON" then the plaintext and the key are:

Plaintext: ATTACKATDAWN
Key: LEMONLEMONLE
Ciphertext: LXFOPVEFRNHR

In order to produce the ciphertext, the first letter of the plaintext is used as a column indicator and the first letter of the key is used as a row indicator, thus the first letter of the ciphertext will

be the letter at the point where the selected column meets the selected row. For our example, following column “A” (due to the first letter of the plaintext) and row “L” (due to the first letter of the key) we find out that the first letter of the ciphertext is “L”. Performing the same operation for the second letter of the plaintext and the second letter of the key we find out that the second ciphertext character is “X”. This process repeats for the entire plaintext, thus generating “LXFOPVEFRNHR”. Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in that row and then using the column's label as the plaintext.

Implementation details

The Vigenère cipher has several Caesar ciphers in sequence with different shift values based on a secret key. For encryption/decryption the keyword is repeated until it matches the length of the plaintext, thus generating the key. The characters that can be found in the alphabet (used for the message and key) should **only be** the uppercase characters A-Z, thus lowercase characters a-z, digits 0-9 or any other ASCII characters **should not** be used. The function(s) encrypting and decrypting the messages should receive as arguments the plain- or cipher-text as well as the keyphrase and should return the result of the operation. They keyphrase will be expanded to match the plaintext or ciphertext length internally, in the corresponding function.

Demo program

In order to evaluate the library you should develop a simple demo program that utilizes the implemented functions. The demo program will prompt the user for a plaintext to be encrypted and a key, in the case of Caesars and Vinegere’s cipher. For each algorithm, the demo program will first encrypt the user input and print the encrypted message. Then, it will decrypt the encrypted message and print the plaintext. **The format has to be the exact following:**

[OTP] input: <user input>
[OTP] encrypted: XXXXXX
[OTP] decrypted: XXXXXX
[Caesars] input: <user input>
[Caesars] key: <a number>
[Caesars] encrypted: XXXXX
[Caesars] decrypted: XXXXX
[Vigenere] input: <user input>
[Vigenere] key: <user input>
[Vigenere] encrypted: XXXX
[Vigenere] decrypted: XXXX

An example execution could be the following.

[OTP] input: secret
[OTP] encrypted: w4&=5Q
[OTP] decrypted: secret

[Caesars] input: hello
[Caesars] key: 4
[Caesars] encrypted: lipps
[Caesars] decrypted: hello
[Vigenere] input: ATTACKATDAWN
[Vigenere] key: LEMON
[Vigenere] encrypted: LXFOPVEFRNHR
[Vigenere] decrypted: ATTACKATDAWN

Important notes

1. You need to submit the "**simple_crypto.h"**, **simple_crypto.c"**, a "**Makefile"** that compiles the library, the "**demoprogram"** that utilizes all the implemented functions and demonstrates their correct usage, and a "**README.txt"** file that explains your implementation and what you didn't implement and why. Please place all these files in a folder named <yourlastnameAM>_assign1, and then compress it as a .zip file that you will upload to eclass. For example: christodoulou2018123456_assign1.zip.
2. The README.txt file is important to submit, and if you have implemented more helper functions or macros explain their functionality in the README.txt file.
3. 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)"
4. This assignment should be implemented using C on Linux-based machines.
5. Follow the steps described above and do an incremental implementation. This will be very helpful in order to debug the various parts before putting them all together in the test file.
6. Submitted code will be tested using plagiarism-detection software.
7. The assignments will be evaluated automatically so the output of the demo program has to follow the format described above.