# COMP2700 Lab 8 – Stream Ciphers and Block Ciphers

The following exercises are adapted from selected problems from Paar & Pelzl's "Understanding Cryptography" (Chapter 2 & 4).

## Lab Environment Setup

For Exercise 1, we will use Python 3 and the PyCryptodome library to perform computations related to modular arithmetic, e.g., computing reminders, multiplicative inverses, etc.

In the following, we assume a linux lab computer is used, but it should work in other OSes if you have Python 3 and pip3 installed.

- If you are using linuxvdi, run the following commands from a terminal to install PyCryptodome:

  ```
  $ pip3 install pycryptodome
  ```

- If you are using your own linux installation, you may need to install pip3 first:

  ```
  $ sudo apt install python3-pip
  ```

  Then install pycryptodome (without using 'sudo')

  ```
  $ pip3 install pycryptodome
  ```

In addition to the pycryptodome library, we will also be using the following two websites for demonstrating AES:

- CrypTool-Online: https://www.cryptool.org/en/cto/aes-step-by-step
- CyberChef: https://gchq.github.io/CyberChef/

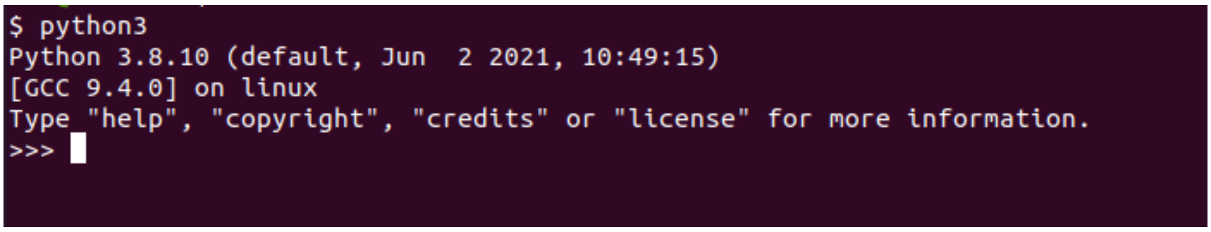
Exercise 5 requires some files to complete; these can be found in the file **lab8_files.zip** on the Wattle page for this lab.

Note that although Exercise 5 is considered an extension exercise, you are recommended to attempt this and review the solution that we will provide later (after the lab sessions end). This will be relevant to the upcoming Assignment 2.

**A brief introduction to PyCryptodome library**. To solve Exercise 1, you may use the 'inverse' function in Crypto.Util.number library of PyCryptodome to help you find the multiplicative inverse modulo m. To use this library, we will run python3 in the interactive mode. In the bash shell run:

$ python3

This will show the python3 interactive prompt:



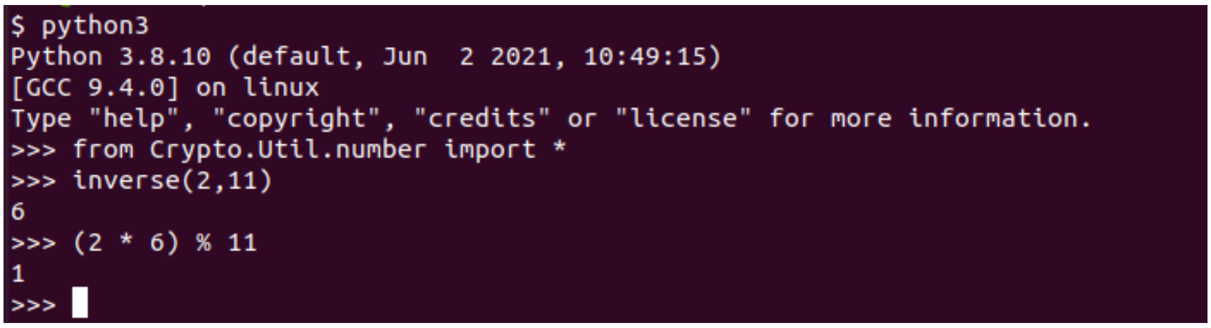
```
$ python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The inverse function is in the Crypto.Util.number, which needs to be imported first, using the following command:

```
from Crypto.Util.number import *
```

This tells python to import all functions from Crypto.Util.number, including the inverse function.

The function 'inverse' takes two arguments: a number we want to find the inverse of, and the modulus. For example, to find $2^{-1} \, mod \, 11$, run 'inverse(2,11)', which gives us 6. You can confirm that 6 is indeed the multiplicative inverse of 2 by computing $2 \times 6 \, mod \, 11$, which gives you 1.

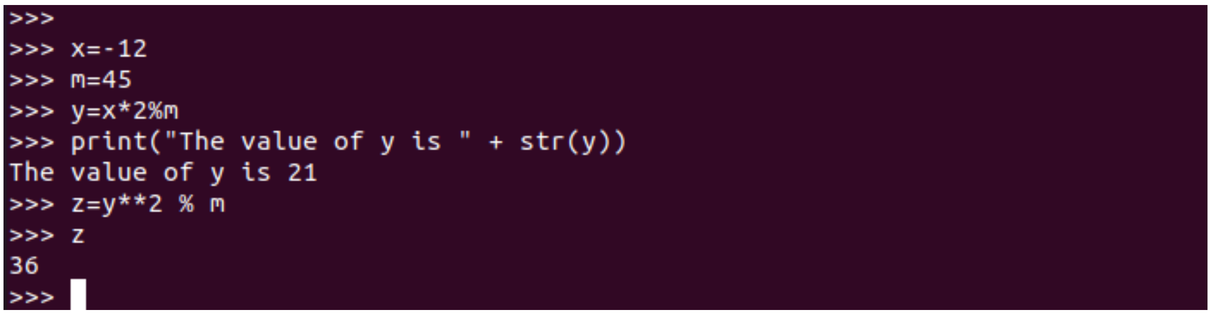


```
$ python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from Crypto.Util.number import *
>>> inverse(2,11)
6
>>> (2 * 6) % 11
1
>>> 
```

For this exercise, you only need to know a few basic arithmetic operators and the concept of variables. The following website provides a quick introduction to these concepts; make sure you go over this introduction quickly if you do not already know python syntax.

https://www.pythoncheatsheet.org

The following is an example of a simple calculation using python, utilising some arithmetic operators and variables.



```
>>>
>>> x=-12
>>> m=45
>>> y=x*2%m
>>> print("The value of y is " + str(y))
The value of y is 21
>>> z=y**2 % m
>>> z
36
>>> 
```

It first assigns -12 to the variable $x$, and 45 to the variable m. It then calculates $x \times 2 \, mod \, m$, and stores the result in variable $y$. The next command prints the value of $y$ – the str function converts integers to strings. It then computes $y^2 \, mod \, m$, and stores its result in $z$.

## Exercise 1.

Consider a PRNG built using the linear congruential generator, with secret seed $S_0$ and secret coefficients $A$ and $B$. The modulus $m = 64283$ is public, so it is known to the attacker. Recall that the linear congruential generator is defined as follows:

$$S_o = \; seed$$
$$S_{i+1} = AS_i + B \; mod \; m$$

where *seed* is a secret seed value.

Suppose the attacker manages to obtain the first three consecutive random numbers generated by this PRNG: $S_1 = 7667, \; S_2 = 50997, \; S_3 = 6447$. Show how the attacker can recover the secrets $S_0$, A and B. Use the PyCryptodome library in python to help you calculate the values of $S_0, A$ and $B$.

## Exercise 2.

*Addition in $GF(2^4)$*: Let $A(x) = x^2 + 1, \; B(x) = x^3 + x^2 + 1$. Compute the following operations in $GF(2^4)$ using the irreducible polynomial $P(x) = x^4 + x + 1$.

a. $A(x) + B(x) \; mod \; P(x)$
b. $A(x) - B(x) \; mod \; P(x)$
c. $A(x) \cdot B(x) \; mod \; P(x)$

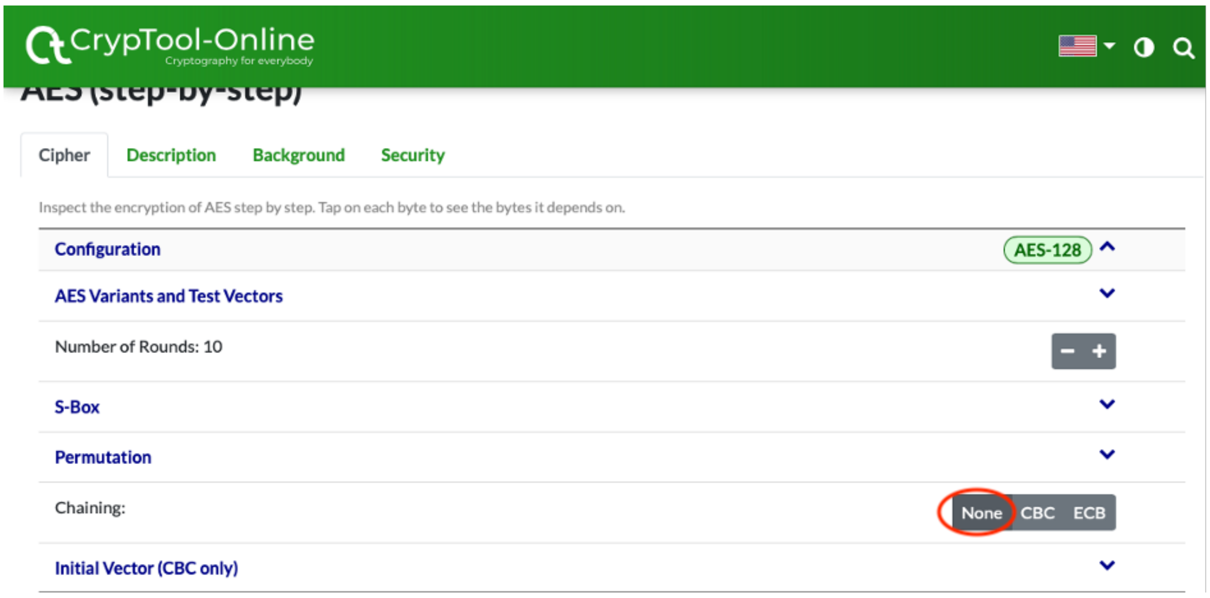
## Exercise 3.

Let $A(x) = x^3 + 1$ and $B(x) = x^6 + x^4 + x$. Compute $A(x) \cdot B^{-1}(x) \; mod \; P(x)$ in $GF(2^8)$ using the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. Then give the representations of $A(x) \cdot B^{-1}(x) \; mod \; P(x)$ as a bit vector and as a HEX value. You may use the following inverse table to help find the inverse $B^{-1}(x) \; mod \; P(x)$.

| | | Y | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| | 0 | 00 | 01 | 8D | F6 | CB | 52 | 7B | D1 | E8 | 4F | 29 | C0 | B0 | E1 | E5 | C7 |
| | 1 | 74 | B4 | AA | 4B | 99 | 2B | 60 | 5F | 58 | 3F | FD | CC | FF | 40 | EE | B2 |
| | 2 | 3A | 6E | 5A | F1 | 55 | 4D | A8 | C9 | C1 | 0A | 98 | 15 | 30 | 44 | A2 | C2 |
| | 3 | 2C | 45 | 92 | 6C | F3 | 39 | 66 | 42 | F2 | 35 | 20 | 6F | 77 | BB | 59 | 19 |
| | 4 | 1D | FE | 37 | 67 | 2D | 31 | F5 | 69 | A7 | 64 | AB | 13 | 54 | 25 | E9 | 09 |
| | 5 | ED | 5C | 05 | CA | 4C | 24 | 87 | BF | 18 | 3E | 22 | F0 | 51 | EC | 61 | 17 |
| | 6 | 16 | 5E | AF | D3 | 49 | A6 | 36 | 43 | F4 | 47 | 91 | DF | 33 | 93 | 21 | 3B |
| | 7 | 79 | B7 | 97 | 85 | 10 | B5 | BA | 3C | B6 | 70 | D0 | 06 | A1 | FA | 81 | 82 |
| X | 8 | 83 | 7E | 7F | 80 | 96 | 73 | BE | 56 | 9B | 9E | 95 | D9 | F7 | 02 | B9 | A4 |
| | 9 | DE | 6A | 32 | 6D | D8 | 8A | 84 | 72 | 2A | 14 | 9F | 88 | F9 | DC | 89 | 9A |
| | A | FB | 7C | 2E | C3 | 8F | B8 | 65 | 48 | 26 | C8 | 12 | 4A | CE | E7 | D2 | 62 |
| | B | 0C | E0 | 1F | EF | 11 | 75 | 78 | 71 | A5 | 8E | 76 | 3D | BD | BC | 86 | 57 |
| | C | 0B | 28 | 2F | A3 | DA | D4 | E4 | 0F | A9 | 27 | 53 | 04 | 1B | FC | AC | E6 |
| | D | 7A | 07 | AE | 63 | C5 | DB | E2 | EA | 94 | 8B | C4 | D5 | 9D | F8 | 90 | 6B |
| | E | B1 | 0D | D6 | EB | C6 | 0E | CF | AD | 08 | 4E | D7 | E3 | 5D | 50 | 1E | B3 |
| | F | 5B | 23 | 38 | 34 | 68 | 46 | 03 | 8C | DD | 9C | 7D | A0 | CD | 1A | 41 | 1C |

**Exercise 4**. In this exercise, we will use the CrypTool Online and CyberChef web apps to check the internals of the AES encryption.

First let's open the CrypTool-Online AES (step-by-step) webpage, and in the 'Configuration' section, under the entry 'Chaining', select None. Leave everything else in this section to their default values.



Inspect the encryption of AES step by step. Tap on each byte to see the bytes it depends on.

Configuration AES-128

AES Variants and Test Vectors

Number of Rounds: 10

S-Box

Permutation

Chaining: [✓ None] CBC ECB

Initial Vector (CBC only)

In the "Key" section, input the following 16 bytes value (in HEX):

00010203 04050607 08090a0b 0c0d0e0f

This will be our encryption key.

In the "Input" section, input the following 16 bytes:

00000000 01010101 02020202 03030303.

This will be our plaintext to be encrypted.

a. Let's start by figuring out the subkeys generated from our encryption key. Use the "Round N" (where N=1..10) section to identify the subkey used in round N. What are the subkeys used in Round 1 and Round 2?
b. Notice that in Round 1, the "input to Round 1" is listed as:

   00010203050407060a0b08090f0e0d0c

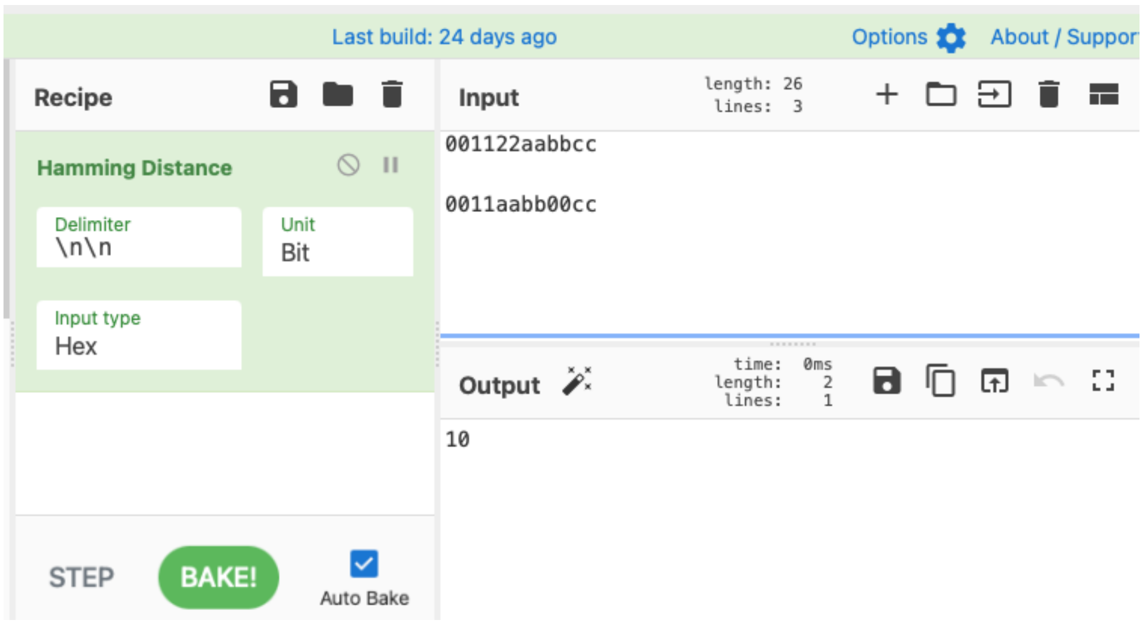
   which is different from our original input above. What is the reason for this?
c. Let's now check the diffusion property of AES after a single round. Recall that the diffusion property measures the number of bit flips in the ciphertext when one bit in the input is flipped. You can do this by comparing the output of Round 1 with the original input, and the output of the same round with the following input (which is the original input, with the first byte changed from 00 to 01):

   01000000 01010101 02020202 03030303

   How many bits are flipped in the ciphertext when you change one bit in the plaintext, when you look at the result of Round 1?

   Note: *given two bit strings of the same length, their* ***Hamming*** *distance is the number of positions for which the symbols at those positions differ. For example, the Hamming distance between 100 and 010 is two, since they differ in first bit (from the left) and the second bit. The measure for diffusion is essentially the Hamming distance between the original ciphertext and the ciphertext after the bit flip in the plaintext. To help you calculate the Hamming distance, you can use the CyberChef website:*
   *in the search bar, search for 'Hamming'.*
   - *In the search bar, search for 'Hamming'.*
   - *Then drag and drop the 'Hamming Distance' entry from the left panel to the 'Recipe' panel.*
   - *In the Recipe panel, make sure you change the 'Unit' to 'Bit', and the 'Input type' to Hex.*
   - *Input the two ciphertexts you want to calculate the Hamming distance for in the Input textbox. Separate them by two newlines.*

     *For example, here's the Hamming distance calculation for two HEX values 001122aabbcc and 0011aabb00cc.*



Delimiter: `\n\n`
Unit: Bit
Input type: Hex

Input (length: 26, lines: 3):

```
001122aabbcc

0011aabb00cc
```

Output (time: 0ms, length: 2, lines: 1):

```
10
```

d. What is the minimum number of rounds required to ensure that, on average, approximately half of the bits in the ciphertext are flipped when one bit in the plaintext is flipped? You can repeat the test you did in 4.c) above, for different rounds. For each round you test, try to vary the position of the bit flip in the input and record the corresponding bit flips in the output, to convince yourself that that round achieves the desired diffusion.

## Extension Exercises (Optional)

### Exercise 5 (*). *[You will need to download some files to complete this exercise; these are provided in the compressed file lab8_files.zip on Wattle]*

In this exercise, we consider a stream cipher built using a linear congruential generator (LCG), similar to the one discussed in Exercise 1. We will use the modulus $m = 64283$. This LCG defines a pseudo-random number generator; we will these random numbers as the key stream, and treat the triple $(S_0, A, B)$ as the key. Each random number produced by this PRNG can be represented by two bytes (since the modulus is 64283 < $2^{16}$). So the keystream is generated two-bytes at a time, following the scheme:

$$S_1 = A \cdot S_0 + B \ mod \ 64283$$
$$S_{i+1} = A \cdot S_i + B \ mod \ 64283$$

Note that $S_0$ is not part of the keystream; it is part of the key.

For example, if the key is $(S_0 = 283, A = 991, B = 1778)$, then the first three random numbers generated are:

25099, 61649, 27087

These will be treated as defining the first 6 bytes of the keystream (represented here in HEX):

620BF0D169CF

(since 25099 = 0x620B, 61649 = 0xF0D1, and 27087 = 0x69CF).

Let $s$ denote the keystream generated using the above LCG scheme with key $K = (S_0, A, B)$. The encryption and the decryption functions for this stream cipher are:

$$e_k(x_i) = x_i \oplus s_i$$
$$d_k(y_i) = y_i \oplus s_i$$

where $s_i, x_i$ and $y_i$ denote, respectively, the i-th bit of $s, x$ and $y$.

You are given an encrypted HTML file (called **doc.enc** – which you can download from the Wattle page for this lab), which was encrypted using this stream cipher with an unknown key. Your task is to decrypt that file to recover the plaintext.

*Hint: use the fact that the original plaintext is an HTML file – which means that the first six characters of the plaintext should contain the HTML tag <html>.*

In your attack, you may find it useful to use python, with the Pycryptodome library, to help you calculate various key elements and perform operations on byte representation of the ciphertext. A few things that may help:

- Crypto.Util.strxor: this has a function called 'strxor' that allows you to compute the result of XOR-ing two byte sequences. This function operates on the 'bytes' data type in python (which is essentially a list of bytes).
- Crypto.Util.number: this contains two functions, long_to_bytes and bytes_to_long, to help you convert a number to a list of bytes and vice versa.
- You can convert a string constant to 'bytes' by prefixing it with a 'b'. For example, you can convert the string 'hello' to bytes by simply writing b'hello'.
- To extract a subsequence from a byte sequence X, you can use the the notation X[m:n], which extracts the bytes from position m up to and including position (n-1). For example, X[0:5] gives you the first 5 bytes in X.

Two python scripts are included. One is **lcgcipher.py**, which is an implementation of the above LCG stream cipher. The other is **findkey_template.py**, which contains skeleton python code that reads the encrypted file into bytes.

If you are not already familiar with the bytes data type, see

https://docs.python.org/3/library/stdtypes.html#binary-sequence-types-bytes-bytearray-memoryview

for more details.

## Exercise 6 (*).

The MixColumn transformation of AES consists of a matrix–vector multiplication in the field $GF(2^8)$ with $P(x) = x^8 + x^4 + x^3 + x + 1$. Let $b = (b_7\,x^7 + b_6x^6 + \cdots + b_1x + b_0)$ be one of the (four) input bytes to the vector–matrix multiplication. Each input byte is multiplied with the constants 01, 02 and 03. Your task is to provide exact equations for computing those three constant multiplications. We denote the result by $d\ = (d_7x^7 + \cdots + d_0)$.

a) Equations for computing the 8 bits of d =01·b.
b) Equations for computing the 8 bits of d =02·b.
c) Equations for computing the 8 bits of d =03·b.

*Note: The AES specification uses "01" to represent the polynomial 1, "02" to represent the polynomial x, and "03" to represent x+1.*

## Exercise 7 (*).

Derive the bit representation for the following round constants within the key schedule: RC[8], RC[9] and RC[10]. Recall that the round constant RC[i] is the polynominal $x^i$.