The Chinese University of Hong Kong

Department of Information Engineering

Summer Workshop – Fun with Information Engineering and Security

Lab 4 – Modern Ciphers

Introduction

Modern ciphers are advanced encryption algorithms used to secure digital data. Unlike traditional ciphers, which often rely on simple substitution or transposition methods, modern ciphers employ complex mathematical techniques to ensure data confidentiality.

In this lab, we will introduce modern ciphers, showcasing advancements in encryption technologies used today. We will introduce some common techniques and schemes used in modern cipher, and work on some hands-on exercises where the encryption scheme is not applied appropriately.

Section 1: Modern Ciphers

Similar to classic ciphers, modern cipher is designed to secure data by transforming readable plaintext into unreadable ciphertext. Compared to classic ciphers, modern ciphers utilize more complicated mathematical functions and longer keys to align with the increased computational power. Modern ciphers can be divided into 2 categories:

Symmetric Ciphers: Uses the same key for both encryption and decryption. Examples include AES and DES. The main types of symmetric ciphers are stream Ciphers and Block Ciphers

Asymmetric Ciphers: Uses a pair of keys—one public and one private. The public key encrypts the data, and the private key decrypts it. Examples include RSA and ECC.

Section 1.1: XOR operator

XOR (Exclusive OR) is an operation used in cryptography. It compares two statements and returns True if they are different; and False if they are the same. The following shows the truth table of "XOR" operations:

XOR

Statement 1	Statement 2	Output
True (1)	True (1)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
False (0)	False (0)	False (0)

*True = 1, False = 0

Consider the following:

0100 XOR 0011

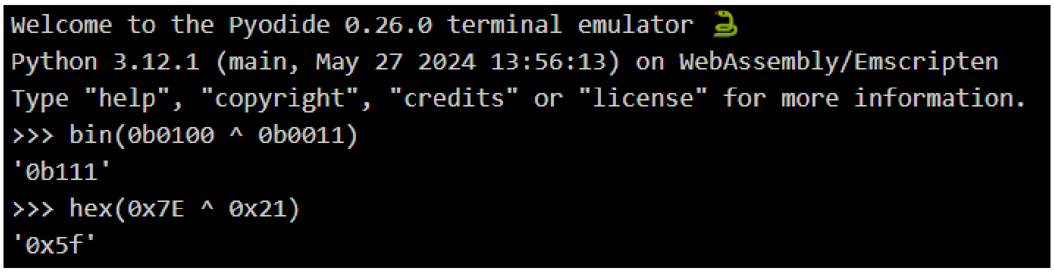
= 0111

Try it!

You may go to https://iesummerworkshop.github.io/pyodide.html. What are the results of the following table? Test the operation on the website using Python to get the result

Statement 1	Operator	Statement 2	Output
1011_2	XOR	0100_2	
0110_2	XOR	1110_2	
$3F_{16}$	XOR	42_{16}	
$2A_{16}$	XOR	$7B_{16}$	

Hints: the XOR operator in Python is `^`, where `0b` declares a binary value and `0x` declares a hexadecimal value. You may use `bin()` and `hex()` to encode the output to binary and hexadecimal values respectively.



```
Welcome to the Pyodide 0.26.0 terminal emulator 🐍
Python 3.12.1 (main, May 27 2024 13:56:13) on WebAssembly/Emscripten
Type "help", "copyright", "credits" or "license" for more information.
>>> bin(0b0100 ^ 0b0011)
'0b111'
>>> hex(0x7E ^ 0x21)
'0x5f'
```

XOR operations are widely used in symmetric key encryption. Each bit of the plaintext combines with the corresponding bit of the key.

Section 1.2: Stream Cipher

In stream cipher, one “letter” (bit) is encrypted at a time. Because each “letter” is masked by a different “letter”, the frequency feature of the underlying plaintext language is not preserved. Compare this with shift cipher and monoalphabetic substitution cipher.

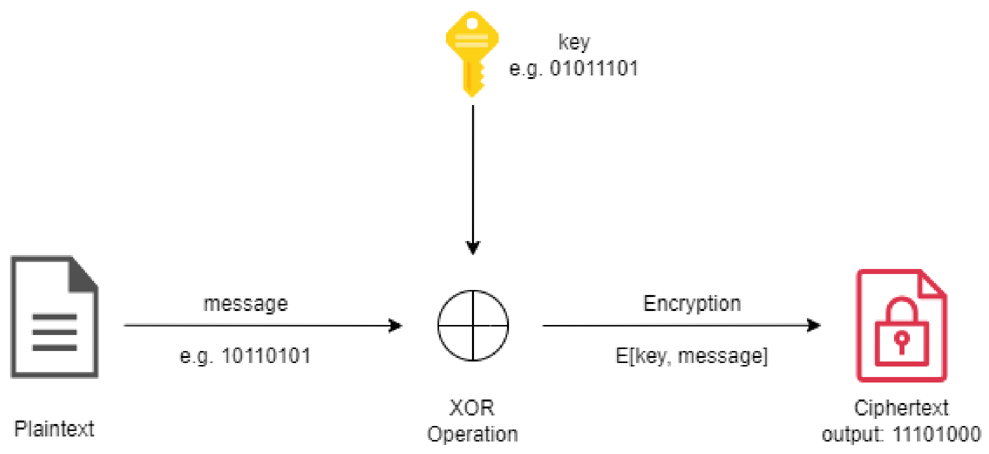
The following examples show how a stream cipher works:

Given the following value:

Plaintext: 10110101_2

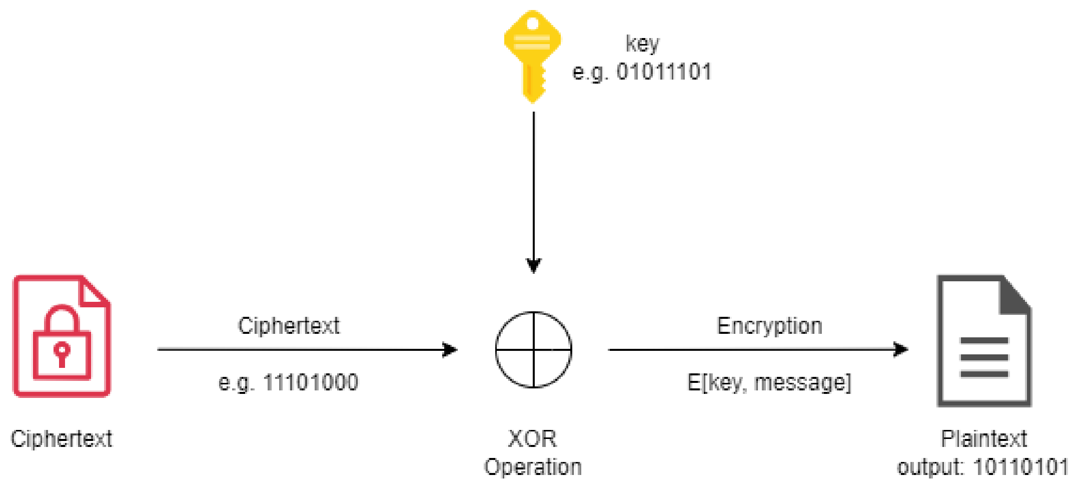
Key: 01011101_2

Encryption



	Most significant bit							Least significant bit
Plaintext	1	0	1	1	0	1	0	1
Key	0	1	0	1	1	1	0	1
Output	1	1	1	0	1	0	0	0

Decryption



	Most significant bit							Least significant bit
Plaintext	1	1	1	0	1	0	0	0
Key	0	1	0	1	1	1	0	1
Output	1	0	1	1	0	1	0	1

How do we get the long key stream? Most of the innovation goes into deriving a cryptographically secure pseudo-random number generator (CSPRNG) that can expand a short key (used as seed) into a long enough key stream. Whether the CSPRNG is secure directly affect the security of the resulting stream cipher.

The following are some common stream ciphers:

RC4: Widely used in protocols like WEP/WPA for wireless security. The key length varies from 8 to 2046 bits. It is known for its simplicity and speed but has vulnerabilities that make it unsuitable for secure applications today.

ChaCha20: A modern stream cipher designed for high security and performance, commonly used in protocols like TLS. The key length is 256 bits. It is highly secure as it is constructed to resist known attacks.

[Optional] Section 1.3: Many time pad

Stream cipher can be vulnerable to attack if the same keystream is used twice or more. If an attacker knows the ciphertext and the corresponding plaintext of one message, they can retrieve the keystream by `plaintext XOR ciphertext` bit by bit. Once the keystream is known, any other message encrypted with the same keystream can be easily decrypted by XOR the ciphertext with the keystream. It is a known-plaintext attack.

For example:

- We have 2 plaintext messages P_1 and P_2
- Assume P_1 and P_2 are encrypted with the same key K

Hence:

$C_1 = P_1 \oplus K$

$C_2 = P_2 \oplus K$

Assume the attacker knows or can guess the content of P_1, the attacker can compute the key by:

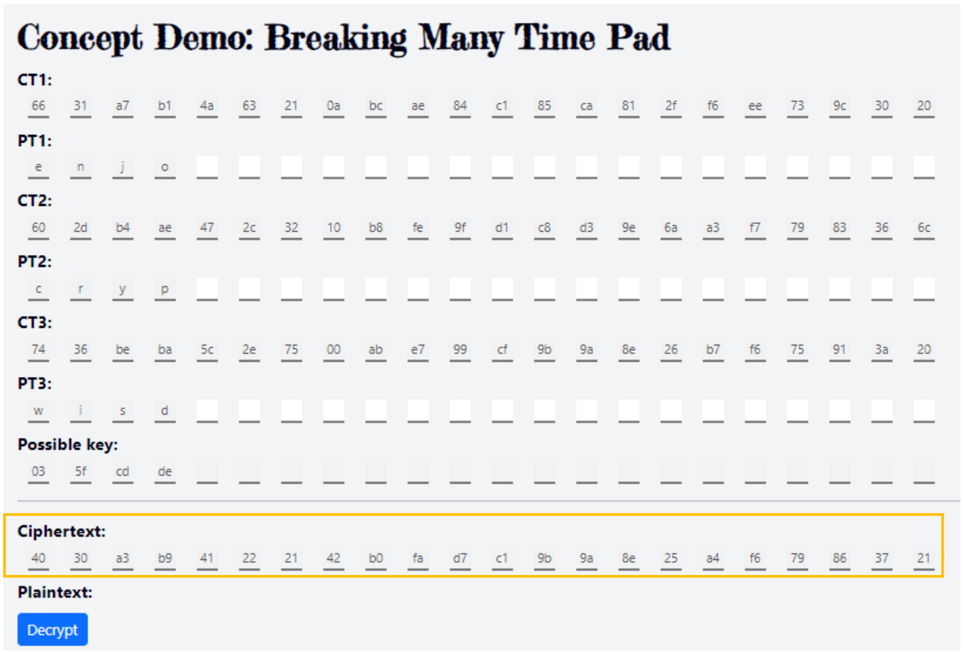
$K = P_1 \oplus C_1$

Then, the attacker can decrypt C_2 with the key:

$P_2 = C_2 \oplus K$

Try it!

Let's try to launch a known plaintext attack. You may go to https://iesummerworkshop.github.io/many-time-pad.html



Concept Demo: Breaking Many Time Pad

CT1:	66	31	a7	b1	4a	63	21	0a	bc	ae	84	c1	85	ca	81	2f	f6	ee	73	9c	30	20
PT1:	e	n	j	o																		
CT2:	60	2d	b4	ae	47	2c	32	10	b8	fe	9f	d1	c8	d3	9e	6a	a3	f7	79	83	36	6c
PT2:	c	r	y	p																		
CT3:	74	36	be	ba	5c	2e	75	00	ab	e7	99	cf	9b	9a	8e	26	b7	f6	75	91	3a	20
PT3:	w	i	s	d																		
Possible key:	03	5f	cd	de																		
Ciphertext:	40	30	a3	b9	41	22	21	42	b0	fa	d7	c1	9b	9a	8e	25	a4	f6	79	86	37	21

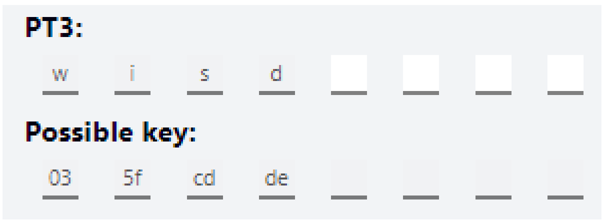
Plaintext:

In this demo page, the following is given:

- 3 completed ciphertext (CT1, CT2, CT3), representing each byte by hexadecimal value
- 3 partial-completed plaintext correspond to the ciphertext (PT1, PT2, PT3), representing each byte by hexadecimal value
- 1 partial-completed key, which is used to encrypt PT1 → CT1, PT2→CT2, PT3→CT3
- 1 set of ciphertext for validating your key [the orange box]

Your task is to guess and recover the key by guessing the possible words in plaintext. For example:

In PT3:



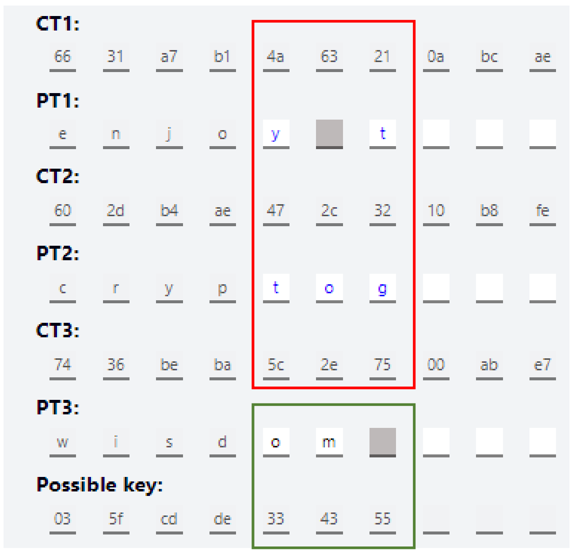
PT3:

w	i	s	d				

Possible key:

03	5f	cd	de				

The plaintext starts with `wisd`, what could be the English word starting with `wisd`? Let's try `wisdom” ”` (adding a space after a word)



CT1:	66	31	a7	b1	4a	63	21	0a	bc	ae
PT1:	e	n	j	o	y		t			
CT2:	60	2d	b4	ae	47	2c	32	10	b8	fe
PT2:	c	r	y	p	t	o	g			
CT3:	74	36	be	ba	5c	2e	75	00	ab	e7
PT3:	w	i	s	d	o	m				
Possible key:	03	5f	cd	de	33	43	55			

After typing om” ”, the demo page computed 2 things for us (computed results are displayed in blue):

1. The possible byte of the key (with XOR operation) [the green box]

Position	Guessing Plaintext	Corresponding Ciphertext	Possible Key
5th byte	o (6f in ASCII)	5c	33
6th byte	m (6d in ASCII)	6d	43
7th byte	“ ” (20 in ASCII)	75	55

2. The corresponding plaintext (PT1 and PT2 in this case), with the key in the previous step [the red box]

Position	Possible Key	CT1	Possible PT1	CT2	Possible CT2
5th byte	33	4a	y	47	t
6th byte	43	64	“ ”	2c	o
7th byte	55	21	t	32	g

- Does the computed plaintext make sense?
- If yes, we may continue guessing the word. Iterate this process until the key is completed.

Next, click on the Decrypt button.

- What is the decrypted message? It should be a meaningful message

Section 1.4: Block Cipher

Block cipher encrypts data in fixed-size blocks. The plaintext will be divided into chunks before processing. If the data is not a multiple of the block size, padding will be added to the final block.

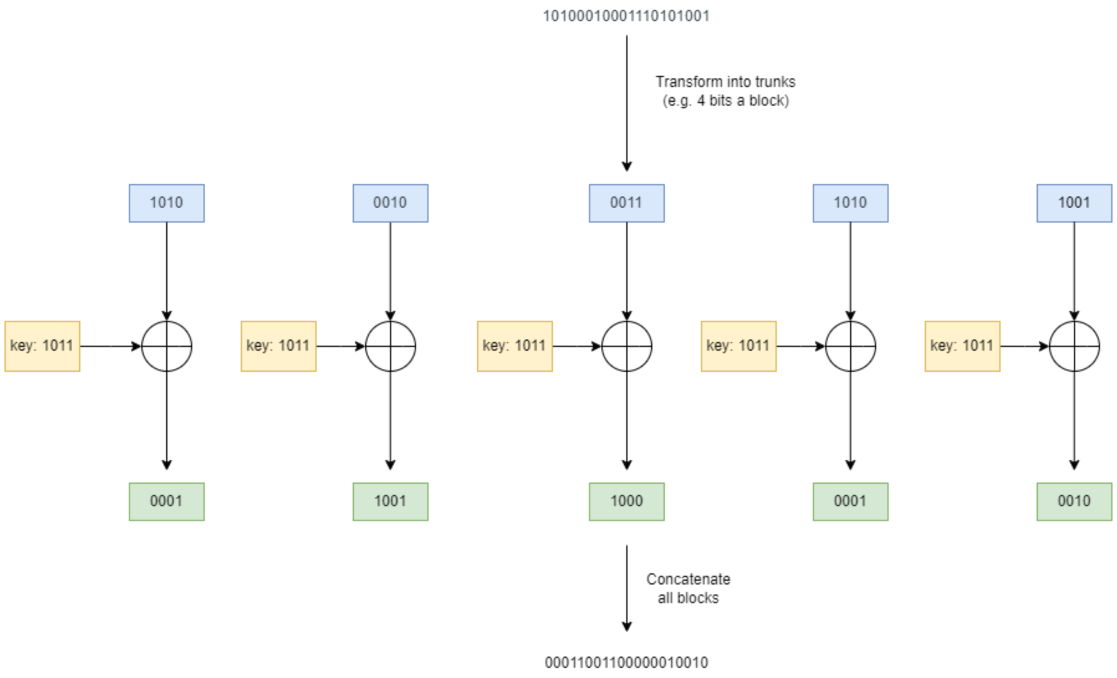
Block cipher masks the underlying frequency feature by making the unit of transformation larger (1 block at a time), rather than encrypting letter by letter (bit by bit). As the name suggests, a block cipher encrypts block by block (which can be seen as a pseudo-random permutation).

<u>Encryption Modes</u>

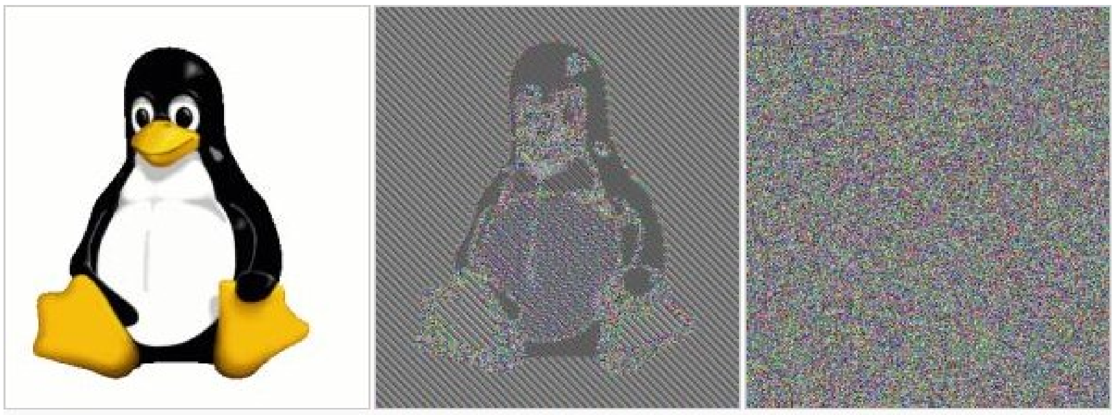
The encryption modes of block ciphers are techniques used to apply block ciphers to encrypt data longer than the block size. The following are two commonly used encryption modes of block ciphers:

1. Electronic Codebook (ECB)

In ECB mode, each block will be encrypted block by block all using the same key. Each block undergoes encryption to produce the ciphertext blocks. Concatenate all the ciphertext blocks to get the final result



As ECB mode relies on simple XOR operation without an initialization vector and chaining, this feature makes ECB mode vulnerable when used in AES. When 2 blocks of plaintext are identical, the ciphertext of the 2 blocks will also be the same.

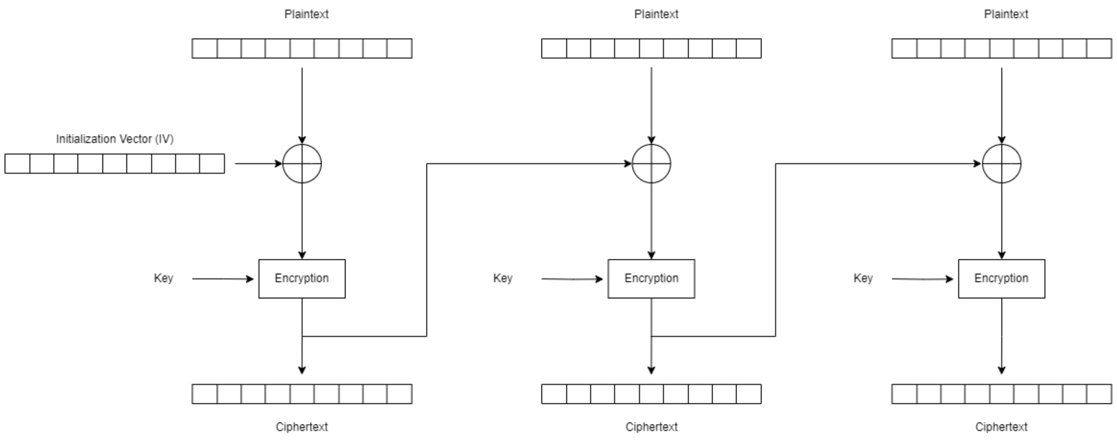


Original image | Encrypted using ECB mode | Modes other than ECB result in pseudo-randomness

Source: https://crypto.stackexchange.com/questions/20941/why-shouldnt-i-use-ecb-encryption

2. Cipher Block Chaining (CBC)

In CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption, introducing dependency between blocks. Initialization Vector (IV) is used as the first block.

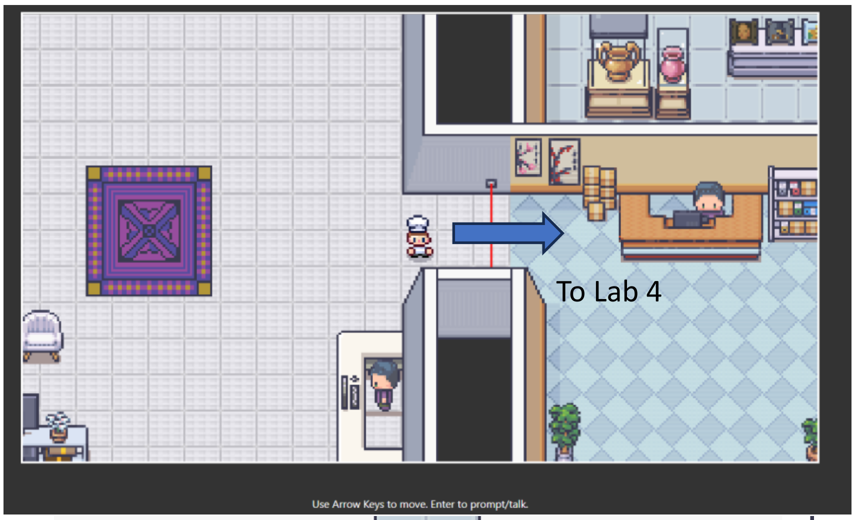


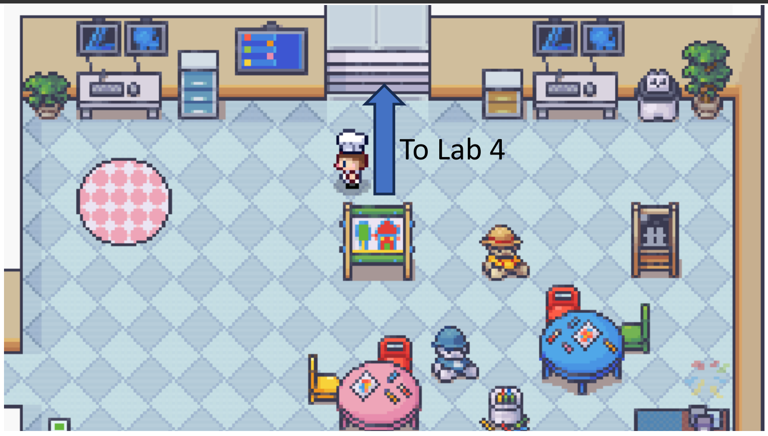
Section 1.5: Stream vs Block cipher

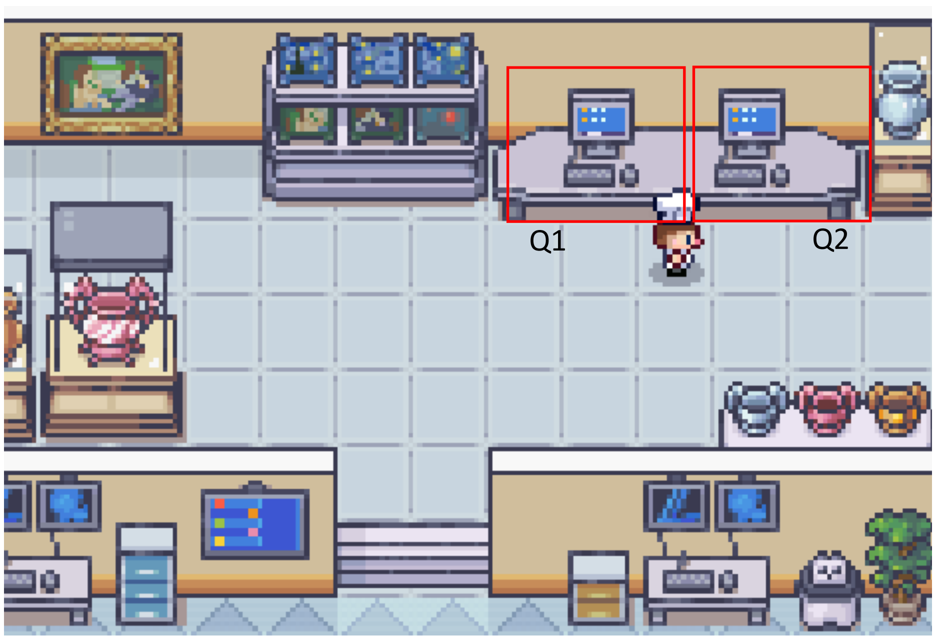
	Stream Ciphers	Block Ciphers
Security	The security relies heavily on the randomness and uniqueness of the keystream. If the same keystream is reused, it can lead to vulnerabilities.	Security can be affected by the mode of operation.
Performance	👍 tend to be faster	Tend to be slower and require more processing power
Complexity	👍 tend to be simpler to implement	Tend more complex to implement and manage

Section 2: Challenge Time

Now, please go to https://iesummerworkshop.github.io/rpg/page.html to enjoy your challenge! Walk in front of the computer and press enter to interact with it. Mark down the key you get!







1. [☆ ☆ ☆] AES decryption

Python supports third-party libraries. By using the simple command pip install, we can install Python packages from the Python Package Index (PyPI) easily. We will be using the 'cryptography' package to decrypt a ciphertext here. Given the key and IV, decrypt the ciphertext using AES CBC mode. Make sure you read the hints and read the API documentation (this is the SOP of most programmers).

2. [☆☆☆☆☆] Breaking LCG PRNG

The LCG PRNG is one of the most popular PRNG, as it is easy to understand and implement. However, it is also cryptographically insecure (and should not be used in a stream cipher).

You should complete the function lcg_attack<(sequence, modulus) to guess what is the next random number based on the given sequence.

The LCG PRNG is defined as:

$$X_{n+1} = (aX_n + c)\ mod\ m$$

Where:

- a is the multiplier
- c is the increment
- m is the modulus
- X_0 is the initial value of the random number sequence, also known as the seed
- X is the sequence of random number

This link might help your understanding of breaking LCG PRNG: https://tailcall.net/posts/cracking-rngs-lcgs/

Interact with this counter to submit the key for lab 3 and lab 4