Quiz 4

111550076 楊子賡

problem 1

Problem 1

LFSR is simply an arrangement of n stages in a row with the last stage, plus any other stages, modulo-two added together and returned to the first stage. An algebraic expression can symbolize this arrangement of stages and tap points called the characteristic polynomial. One kind of characteristic polynomial called primitive polynomials over GF(2), the field with two elements 0, 1, can be used for pseudorandom bit generation to let linear-feedback shift register (LFSR) with maximum cycle length.

a) Is $x^8 + x^4 + x^3 + x^2 + 1$ a primitive polynomial?

b) What is the maximum cycle length generated by $x^8 + x^4 + x^3 + x^2 + 1$?

c) Are all irreducible polynomials primitive polynomials?

a) Yes, because It can't be factorized anymore.

b) The answer 2^8 - 1 = 255 is the maximum cycle length for an LFSR with a primitive polynomial of degree 8. This means the LFSR can produce a sequence of 255 unique states before repeating.

c) not all irreducible polynomials are primitive. For instance, if we work with K=GF(2), then polynomial P= $X^8 + X^4 + X^3 + X + 1$ is irreducible, but not primitive (there are 255=2^8–1 elements modulo P; but the subgroup generated by X only contains 51 of them).

Problem 2

Given the plaintext:

ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRAN

SCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCO

MPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUET

OBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMU

CHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLT

HATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSI

TYINTHEFIRSTPLACE

a) Please use $x^8 + x^4 + x^3 + x^2 + 1$ as a characteristic polynomial to write a Python program to encrypt the following plaintext message with the initial key 00000001, then decrypt it to see if your encryption is correct.

b) Due to the property of ASCII coding the ASCII A to Z, the MSB of each byte will be zero (left most bit); therefore, every 8 bits will reveal 1 bit of random number (i.e. keystream); if it is possible to find out the characteristic polynomial of a system by solving of linear equations?

c) **Extra credit:** Write a linear equations program solving program to find the characteristic polynomial for this encryption with initial 00000001.

a) Using python `pylsfr` package to easily get the last bit of each round. Save it as 'keystream' to use.

```
# to run the code, you should type:
pip install pylfsr
#in terminal
```

```
from pylfsr import LFSR
```

```
def generate_keystream(L, length):
    keystream = []
    for _ in range(length):
        keystream.append(L.state[-1])
        L.next()
    return keystream

state = [1,0,0,0,0,0,0,0]  # Initial key '00000001'
taps = [8,4,3,2]  # Polynomial x^8 + x^4 + x^3 + x^2 + 1
L = LFSR(initstate=state, fpoly=taps)
plaintext_length = len(plaintext) * 8
# Calculate length of keystream needed (8 bits for each char)
keystream = generate_keystream(L, plaintext_length)
```

In stream ciphers, encryption and decryption use the same process because they both involve XORing the data with a keystream. XORing a message with the same keystream twice returns the original message, making the process symmetric and efficient.

```
def encrypt_decrypt_with_keystream(input_text, keystream):
    output_text = ''
    keystream_index = 0  # Index to track position in keystream

    for char in input_text:
        encrypted_chars = []
        for bit in format(ord(char), '08b'):  # Convert each cha
            keystream_bit = keystream[keystream_index]
            keystream_index += 1  # Move to next bit in keystrea
            encrypted_bit = str(int(bit) ^ keystream_bit)
            # XOR with corresponding bit of keystream
            encrypted_chars.append(encrypted_bit)
        encrypted_char = chr(int(''.join(encrypted_chars), 2))
        # Convert back to character
```

```
        output_text += encrypted_char

    return output_text
```

```
# main
ciphertext = encrypt_decrypt_with_keystream(plaintext, keystrear
print("Ciphertext:", ciphertext)

decrypted_text = encrypt_decrypt_with_keystream(ciphertext, keys
print("Decrypted text:", decrypted_text)

print("keystream:",''.join(str(i) for i in keystream))
```

Ciphertext:

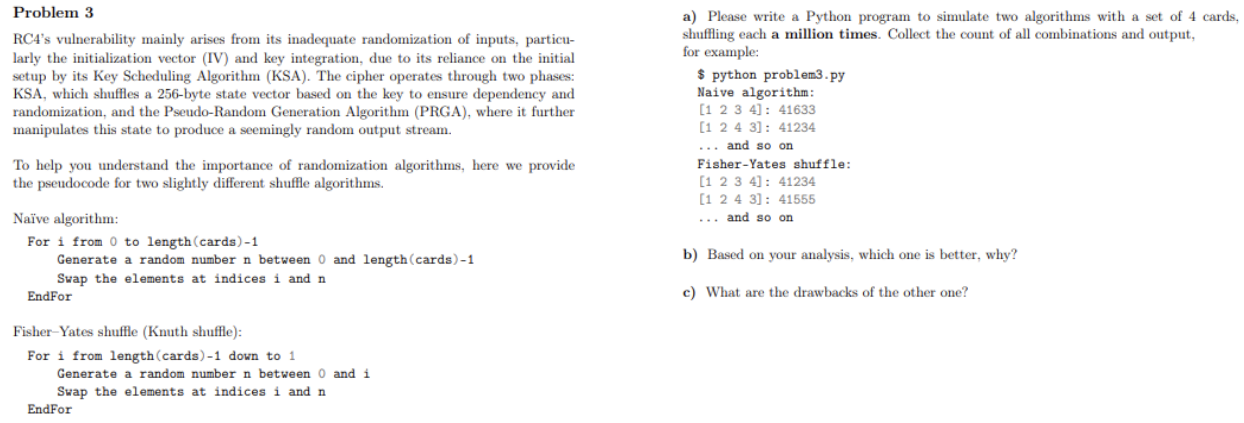
bý¥4ïÝ=aÿû[,8LrR«Tó
Ë
Âë-éÔfÅÏ¢ÃÀ?5gCþ@Ð\Ë
ªÌzîÐAÖ-¼;|yQRFù¤½(þái#ÔD`ÇFNË¼R-Ii~¸«½êÊ~T"¢¢uiUAtC¾WµàX+®æñ
i%f®£ÿâèØÛ$1Ïtd;K³I¶Ví¿ÒüÌ;5þ
xI»Y
±mýeÒ½c:ä\Ù²ñÅæ!ÂÓË2ÀtÙð®+Õõ<.çg:²
v÷q¦(S}ôP ÖGð>U+¼s
¤KòªÓÏ=kïíS?3@rO

b)

Yes, it is possible to determine the characteristic polynomial of a Linear Feedback Shift Register (LFSR) system by solving linear equations. This method takes advantage of the property of ASCII encoding where the most significant bit (MSB) of each byte representing letters 'A' to 'Z' is zero. In the context of encryption, if a plaintext composed entirely of these letters is encrypted using an LFSR-based stream cipher, the MSB of the keystream can be directly inferred from the MSB of the ciphertext.

By collecting enough bits from the inferred keystream (ideally as many as the length of the LFSR), one can set up a system of linear equations that represent the feedback function of the LFSR. Solving this system, typically using methods like Gaussian elimination, can reveal the characteristic polynomial of the LFSR, which dictates its feedback taps and overall behavior. This approach relies on having a sufficient sequence of keystream bits and accurate assumptions about the plaintext and ciphertext relationship.

Problem 3



Problem 3

RC4's vulnerability mainly arises from its inadequate randomization of inputs, particularly the initialization vector (IV) and key integration, due to its reliance on the initial setup by its Key Scheduling Algorithm (KSA). The cipher operates through two phases: KSA, which shuffles a 256-byte state vector based on the key to ensure dependency and randomization, and the Pseudo-Random Generation Algorithm (PRGA), where it further manipulates this state to produce a seemingly random output stream.

To help you understand the importance of randomization algorithms, here we provide the pseudocode for two slightly different shuffle algorithms.

Naïve algorithm:

```
For i from 0 to length(cards)-1
    Generate a random number n between 0 and length(cards)-1
    Swap the elements at indices i and n
EndFor
```

Fisher–Yates shuffle (Knuth shuffle):

```
For i from length(cards)-1 down to 1
    Generate a random number n between 0 and i
    Swap the elements at indices i and n
EndFor
```

a) Please write a Python program to simulate two algorithms with a set of 4 cards, shuffling each **a million times**. Collect the count of all combinations and output, for example:

```
$ python problem3.py
Naive algorithm:
[1 2 3 4]: 41633
[1 2 4 3]: 41234
... and so on
Fisher-Yates shuffle:
[1 2 3 4]: 41234
[1 2 4 3]: 41555
... and so on
```

b) Based on your analysis, which one is better, why?

c) What are the drawbacks of the other one?

a) b)

```
# to run the code, you should type:
pip install random
pip install collections
pip install numpy
#in terminal
```

two algorithm:

```
def naive_shuffle(cards):
    for i in range(len(cards)):
        n = random.randint(0, len(cards) - 1)
        cards[i], cards[n] = cards[n], cards[i]
```

```
def fisher_yates_shuffle(cards):
    for i in range(len(cards)-1, 0, -1):
        n = random.randint(0, i)
        cards[i], cards[n] = cards[n], cards[i]
```

simulate:

```
def simulate_shuffles(num_simulations, num_cards=4):
    naive_counts = defaultdict(int)
    fy_counts = defaultdict(int)
    card_set = list(range(1, num_cards+1))
    for _ in range(num_simulations):
        naive_cards = card_set[:]
        naive_shuffle(naive_cards)
        naive_counts[tuple(naive_cards)] += 1
        fy_cards = card_set[:]
        fisher_yates_shuffle(fy_cards)
        fy_counts[tuple(fy_cards)] += 1
    return naive_counts, fy_counts
```

To determine which one is better, I write a simple analyze to calculate the mean and standard deviation of the occurrence counts for card combinations produced by a shuffle algorithm, indicating the uniformity of the distribution.

```
def analyze_results(counts):
    values = np.array(list(counts.values()))
    mean = np.mean(values)
    std_dev = np.std(values)
    return mean, std_dev
```

```
#main
num_simulations = 1000000
naive_counts, fy_counts = simulate_shuffles(num_simulations)
naive_mean, naive_std = analyze_results(naive_counts)
```

```
fy_mean, fy_std = analyze_results(fy_counts)
print("Naive algorithm:")
for key, value in sorted(naive_counts.items()):
  print(f"{key}: {value}")

print("\nFisher–Yates shuffle:")
for key, value in sorted(fy_counts.items()):
  print(f"{key}: {value}")

print(f"Naive Shuffle: Mean = {naive_mean}, Std Dev = {naive_st
print(f"Fisher-Yates Shuffle: Mean = {fy_mean}, Std Dev = {fy_s

if fy_std < naive_std:
  print("Fisher-Yates shuffle is better: more uniform distribut
else:
  print("Naive shuffle is better: more uniform distribution.")
```

result:

```
(4, 2, 3, 1): 41577
(4, 3, 1, 2): 41576
(4, 3, 2, 1): 41498
Naive Shuffle: Mean = 41666.666666666664, Std Dev = 7288.38774848198
Fisher-Yates Shuffle: Mean = 41666.666666666664, Std Dev = 134.02912204276933
Fisher-Yates shuffle is better: more uniform distribution.
```

```
Naive Shuffle: Mean = 41666.666666666664, Std Dev = 7125.61777828577
Fisher-Yates Shuffle: Mean = 41666.666666666664, Std Dev = 157.7954019890595
Fisher-Yates shuffle is better: more uniform distribution.
```

```
Naive Shuffle: Mean = 41666.666666666664, Std Dev = 7187.210253096971
Fisher-Yates Shuffle: Mean = 41666.666666666664, Std Dev = 151.9201178982633
Fisher-Yates shuffle is better: more uniform distribution.
```

Based on conducting the simulation three times, it has been observed that the `Fisher-Yates shuffle algorithm` consistently performs better, indicating a more uniform distribution of card combinations compared to the Naive shuffle algorithm.

c)

The main drawback of the Naive shuffle algorithm compared to the Fisher-Yates shuffle is that it can introduce bias into the shuffle, resulting in a non-uniform distribution of card combinations. This bias arises because the Naive shuffle allows a card to be swapped with itself, altering the probabilities of different combinations in an unequal manner. This means that certain card sequences can become more likely than others, which is undesirable in applications requiring fair and random shuffling, such as in cryptographic systems or card games.