MTH 329 - Cryptography - Assignment 3

Purva Parmar

20181081

Permutation Cipher

A Permutation Cipher, at its core, simply rearranges characters of the plaintext.

Encryption and Decryption

Let the key be K of length n, presented in Cyclic Permutation notation. The cipher encryption is then:

- Pad the plaintext with any appropriate character if its size is not divisible by n.
- Break the plaintext into blocks of size n each.
- Use the key K to permute each block.
- Join the permuted blocks to form the ciphertext.

For a simple example, suppose the plaintext is `abcdefghi` and the key is `(4, 1, 0, 3, 2)`. The encryption procedure is then:

- The size of the plaintext is 9 while that of the key is 5. So, we pad the plaintext with an `x` to make it `abcdefghix`.
- We break the plaintext into blocks of size 5: `abcde` and `fghix`.
- Permute each block with the key. The ciphertext blocks are:

Plaintext	Ciphertext
abcde	daecb
fghix	ifxhg

- So, the ciphertext is `daecbifxhg`.

For decryption, we follow the encryption process with the key reversed.

Cryptanalysis

For a key of size `n`, there are `n!` possible permutations, if all characters in each block are distinct. Searching through the entire keyspace with a bruteforce attack is not really feasible.

We could try a known plaintext and chosen plaintext attack, if possible.

Finding Key Length

Suppose the length of plaintext is N. Factorize N. The key size is one of these factors.

For example, if length of plaintext is 30, then we see that factors of 30 are 2, 3, 5, 6, 10, 15, 30. We could test all of these possibilities, starting with a reasonable guess. For example, a reasonable guess is that the key size is 10. We could test that first.

Known Plaintext Attack

We have a plaintext P and the corresponding ciphertext C. Break the ciphertext into blocks of guessed key size. For simplicity, let's assume we have guessed the correct key size. Consider the example:

The plaintext block is `linux` and the ciphertext block is `ulxni`. The key K is `(4, 1, 0, 3, 2)`, but we don't know that yet.

We start counting indexes from 0. The decryption procedure.

- `l` goes to Index `1`. At this index, the plaintext character is `i`.
- `i` goes to Index `4`. At this index, the plaintext character is `x`.
- `x` goes to Index `2`. At this index, the plaintext character is `n`.
- `n` goes to Index `3`. At this index, the plaintext character is `u`.
- `u` goes to Index `0`. At this index, the plaintext character is `l`.

We obtained a cycle path `(1, 4, 2, 3, 0)`. Reverse this, and we get `(0, 3, 2, 4, 1)`. This is in cycle notation, so we can rotate it to `(4, 1, 0, 3, 2)`, and this was indeed the key that we started with. We can use this key to encrypt any other blocks and check that they indeed give the correct ciphertext block. The known plaintext attack is complete.

Chosen Plaintext Attack

We can use the same procedure as the known plaintext attack. We generate a plaintext with all unique characters in sequential order, get the ciphertext, and follow the procedure of the known plaintext attack. Here, we don't just hope that we will get a block with all unique characters, we are instead providing such a block for it to work.

For example, if the key size is 4, we provide a plaintext with the characters `abcd`. This makes the indexing easier to work with (for humans).

Another procedure is to keep all characters except one same and check where the one unique character is shifted to. For example, if key length is 4, we pass 4 plaintext blocks: `baaa`, `abaa`, `aaba` and `aaab`. Then, we check where the `b` character is shifted to. This method is simpler while working by hand. Since we are coding things anyway, I follow the first method.

Implementation in Python

Permutation Cipher Class

We create a class and associated methods, to simplify the entire implementation. This class would then act as our "cipher machine", and we would be able to encrypt and decrypt stuff without knowing the key. This is useful for known and chosen plaintext attacks.

The class program is saved in a file named `PermutationCipher.py` (becomes important when importing to other programs).

```
from random import shuffle

class PermutationCipher:
    """
    Permutation Cipher and the associated encryption/decryption functions
    """

    def __init__(self, key=[], keylength=5):
```

```
        """Initialize cipher with either a provided key or keylength

        Args:
            key (list, optional): Key as a list of integers, in cycle notation. Defaults to
[].
            keylength (int, optional): Length of key to generate. Defaults to 5.
        """

        if key:
            self.KEY = key
        else:
            self.KEY = list(range(keylength))

            # Shuffle this range
            shuffle(self.KEY)

    def pad(self, plaintext):
        """Pad plaintext so that all blocks are complete and of the same size

        Args:
            plaintext (str): The plaintext to be

        Returns:
            str: Padded Plaintext
        """

        keylength = len(self.KEY)
        remainder = len(plaintext) % keylength

        if remainder == 0:
            return plaintext

        plaintext += 'x' * (keylength - remainder)
        return plaintext

    def process_text(self, text):
        """Process text: Perform padding and split into blocks

        Args:
            text (str): The text string to be processed

        Returns:
            list: List containing the blocks of text
        """
        keylength = len(self.KEY)
        text = self.pad(text)

        # Split text into blocks
        # Example: "ABCDEF" with keylength=2 gets split as [['A', 'B'], ['C', 'D'], ['E',
'F']]
        blocks = [text[i : i + keylength] for i in range(0, len(text), keylength)]

        return blocks

    def permute(self, text_blocks, key):
        """Permute text in each text block as per the key

        Args:
```

```
            text_blocks (list): List containing blocks of text
            key (list): The key to use for permutation, in cyclic notation

        Returns:
            str: The permuted text
        """
        keylength = len(key)

        permuted_blocks = []
        for block in text_blocks:
            # Initialize permuted block as original block
            pblock = list(block)

            for i in range(keylength):
                # Character at position KEY[i] goes to KEY[i + 1]
                # Modulus with keylength is taken for indexes
                pblock[key[i % keylength]] = block[key[(i + 1) % keylength]]

            permuted_blocks.append(''.join(pblock))
            permuted_text = ''.join(permuted_blocks)

        return permuted_text

    def encrypt(self, plaintext):
        """Encrypt a plaintext with the permutation cipher key

        Args:
            plaintext (str): The plaintext, without any spaces and punctuation

        Returns:
            str: The ciphertext
        """
        plaintext_blocks = self.process_text(plaintext)
        ciphertext = self.permute(plaintext_blocks, self.KEY)
        return ciphertext

    def decrypt(self, ciphertext):
        """Decrypt a ciphertext with the permutation cipher key

        Args:
            ciphertext (str): The ciphertext, without any spaces and punctuation

        Returns:
            str: The plaintext, with the trailing padded letters stripped
        """
        # Perform the encryption steps with reversed key
        # and strip off trailing 'x' letters
        ciphertext_blocks = self.process_text(ciphertext)
        plaintext = self.permute(ciphertext_blocks, self.KEY[::-1]).rstrip('x')
        return plaintext

    def get_key(self):
        """Get the cipher Key

        Returns:
            list: List containing integer entries as the key
        """
        return self.KEY
```

We can initialize the cipher by either providing our own key, or letting it generate a key itself given a keylength.

Here's an example:

```
>>> from PermutationCipher import PermutationCipher
>>> pc = PermutationCipher(keylength=4)
>>> plaintext = "hellothere"
>>> ciphertext = pc.encrypt(plaintext)
>>> print(ciphertext)
lheleothxrex
>>> print(pc.decrypt(ciphertext))
hellothere
```

Known Plaintext Attack

Let the plaintext be `thedarksideoftheforceisapathwaytomanyabilitiessomeconsidertobeunnatural`.

```
In [1]:  from PermutationCipher import PermutationCipher
         pc = PermutationCipher(keylength=7)
```

```
In [2]:  plaintext =
"thedarksideoftheforceisapathwaytomanyabilitiessomeconsidertobeunnatural"
         ciphertext = pc.encrypt(plaintext)
         keylength = 7
         print(f"{ciphertext = }")

Out [2]: ciphertext =
'redkthafdetsiocfoehertaphisamytawaolabinyioesmtisiondecsetouerbratannuxxxxlxx'
```

```
In [3]: def known_plaintext_attack(plaintext, ciphertext, keylength):

            # Split into blocks
            blocks = {plaintext[i:i+keylength]: ciphertext[i:i+keylength] for i in range(0,
len(plaintext), keylength)}

            # Find blocks with all unique characters
            unique_blocks = {pt: blocks[pt] for pt in blocks if len(pt) == len(set(pt))}

            if not unique_blocks:
                print("No unique blocks")
                return []

            # Take the first unique block and corresponding ciphertext
            pt = list(unique_blocks.keys())[0]
            ct = unique_blocks[pt]

            probable_key = list(range(keylength))
```

```
            # The attack
            current_index = 0
            for i in range(keylength):
                probable_key[i] = ct.index(pt[current_index])
                current_index = probable_key[i]

            # Reverse the key
            probable_key = probable_key[::-1]

            return probable_key
```

```
In [4]:  obtained_key = known_plaintext_attack(plaintext, ciphertext, keylength)
         obtained_key


Out [4]: [0, 5, 1, 2, 3, 6, 4]
```

Now, we initialize a new PermutationCipher object and test this new key, to see if it works.

```
In [5]:  testpc = PermutationCipher(key=obtained_key)
         testpc.decrypt(ciphertext)

Out [5]: 'thedarksideoftheforceisapathwaytomanyabilitiessomeconsidertobeunnatural'
```

So, the new key indeed works! We can check that it is equivalent to the original key:

```
In [6]:  # Original Key
         pc.get_key()

Out [6]: [0, 5, 1, 2, 3, 6, 4]
```

So, our known plaintext attack was successful!

Chosen Plaintext Attack

We continue with the functions and variables as defined in the Known Plaintext Attack.

```
In [7]: def chosen_plaintext_attack(PermutationCipher, keylength):
            # Generate a block of plaintext with all unique characters
            pt = ''.join([chr(i + 65) for i in range(keylength)])
            ct = PermutationCipher.encrypt(pt)

            return known_plaintext_attack(pt, ct, keylength)
```

As we can see, it generates a block of unique characters of the size of the key. For example, the key size here is 7. So, the chosen plaintext is `ABCDEFG`. It then passes this plaintext into the function for the known plaintext attack. Since our block has all unique characters, the attack is guaranteed to work, if the guessed key size is correct.

```
In [8]:  chosen_plaintext_attack(pc, keylength)

Out [8]: [0, 5, 1, 2, 3, 6, 4]
```

As we can see, it also returns the correct key. The chosen plaintext attack is successful as well.

Possible Improvements

As any code, this one isn't perfect either. Here are some improvements that could be done to the code:

- Allow key to be mentioned in general cyclic notation with multiple cycles in a key.
- Write a known plaintext attack that doesn't rely on only one block with unique characters. This would involve checking multiple blocks at a time, which would take a lot of thinking to implement.

Code Files

All code files can be found on my GitHub repository:

https://github.com/TheReconPilot/MTH329-Cryptography