CSE 350 Network Security
Programming Assignment Number 1

Akshat Saini 2020019 Nakul Thureja 2020528

Project Number = (0019 + 0528) mod 3 = 1

Project 1: Encryption & decryption using poly-alphabetic substitution of the kind discussed in class. Then develop the software to launch a brute-force attack to discover the key. Assume that the key length is known, and it is 4

Notation used:
Key = k0 k1 k2 k3 (length = 4)
Plaintext = p0 p1 p2 ….. pn-1 (length = n)
Ciphertext = c0 c1 c2 …. cm-1 (length = m)

Input:
A file named plaintext_input.txt which contains all the plaintexts in lowercase letters without any spaces in between one text.
A string of 4 lowercase letters which would be the key used for encryption/decryption.

Output:
A file named brute_force_output.txt which contains the discovered key through the attack and the resultant plaintexts after using the key to decode.
Visual output on the terminal to check if the encryption and decryption functions run properly.

Command to run:
python3 A1_2020019_2020528 [input file name] [key]
For e.g. - python3 A1_2020019_2020528 plaintext_input.txt trdg

System Designed

The system designed has 2 major functions, namely **encrypt and decrypt**, which perform **poly-alphabetic substitution cipher** with a **fixed key length of 4.** Along with these two, there is also a function named **Hash which is responsible for hashing the text.**

1. Encrypt

The encrypt function is responsible for using the key to encrypt the raw messages using a poly-alphabetic substitution cipher.

```
def encrypt(message, key):
    # Encrypts the plaintext using poly-alphabetic substitution cipher
    plaintext = message + Hash(message)
    encoding_key = key_extender(key, len(plaintext))
    ciphertext = ""
    for i in range(len(plaintext)):
        char = (value_of_char(plaintext[i]) + value_of_char(encoding_key[i])) % 26
        ciphertext += char_of_value(char)
    return ciphertext
```

To encrypt, we first **generate a new plaintext** of the format *p = (string, Hash(string))* by **concatenating the message and the hash of the message**. Here the hash is generated by the function Hash(), which will be explained later.
After this, we **expand the key such that the new key is equal in length with the plaintext** and of the form *k0 k1 k2 k3 k0 k1 k2 k3 k0 k1 k2 k3 k1 …….*
Finally, for encryption, we use the following function to calculate and return the ciphertext -
((p0 + k0) mod26) ((p1 + k1) mod26) ((p2 + k2) mod26) ((p3 + k3) mod26) ((p4 + k0) mod26) ((p5 + k1) mod26) …… till pn-1

2. Decrypt

The decrypt function is responsible for using the key to decrypt the ciphertext using a poly-alphabetic substitution cipher.

```
def decrypt(ciphertext, key):
    # Decrypts the ciphertext using poly-alphabetic substitution cipher
    encoding_key = key_extender(key, len(ciphertext))
    plaintext = ""
    for i in range(len(ciphertext)):
        char = (value_of_char(ciphertext[i]) - value_of_char(encoding_key[i]) + 26) % 26
        plaintext += char_of_value(char)
    return plaintext_validity_check(plaintext)
```

To decrypt, we first **expand the key such that the new key is equal in length with the plaintext** and of the form *k0 k1 k2 k3 k0 k1 k2 k3 k0 k1 k2 k3 k1 …….*
Finally, for decryption, we use the following function to reverse the encryption method -
((c0 - k0 + 26) mod26) ((c1 - k1 + 26) mod26) ((c2 - k2 + 26) mod26) ((c3 - k3 + 26) mod26) ((c4 - k0 + 26) mod26) ((c5 - k1 + 26) mod26) …… till cm-1
After this, we send the plaintext received and send it to the plaintext_validity_check function to check if the plaintext received is recognizable.

3. Plaintext Validity Check

This function is responsible for checking if the plaintext achieved is in the recognizable form or not.

```python
def plaintext_validity_check(plaintext):
    # Checks if the plaintext is valid
    index = len(plaintext) - 256//4
    message = plaintext[:index]
    hash = plaintext[index:]
    if Hash(message) == hash:
        return message
    else:
        return "Invalid plaintext"
```

To check if the plaintext is in recognizable form, we first remove the hash that was added previously during the encryption phase and **divide the text into two parts, the message and the hash.** Now we **hash the message and compare it with the original** hash. If both match, we know that we have decoded the plaintext correctly and the message is in recognizable form. If they do not match, the key would have been incorrect, and the decoded message would be just gibberish. **For the successful case, we have simply returned the message, while for the unsuccessful message, we have returned a string "Invalid plaintext".**

4. Hash

This function is responsible for hashing the text.

```python
def Hash(message):
    # Hashes the plaintext using the SHA-256 algorithm
    text = hashlib.sha256()
    text.update(message.encode())
    actual_hash = text.hexdigest()
    mod_hash = ''
    for i in actual_hash:
        if character_set.find(i) == -1:
            mod_hash += character_set[int(i)-1]
        else:
            mod_hash += i
    return mod_hash
```

We have used the implementation of **SHA256 from the *hashlib* library.** The hash generated by the library is **alphanumeric.** To encrypt using poly-alphabetic substitution, the hash should be in the character set ('a' to 'z'), so we have **replaced the numbers with their corresponding character in the alphabet.**

Properties of SHA256 hash used in other functions: Length of Hash is 256/4 = 64

5. Some helper functions

```python
def char_of_value(value):
    # Returns the character of the value
    return chr(value + ord('a'))

def value_of_char(char):
    # Returns the value of the character
    return ord(char) - ord('a')
```

```python
def key_extender(key, size):
    # Extends the key to the size of the plaintext
    while len(key) < size:
        key += key
    return key[:size]
```

char_of_value - Returns the English letter corresponding to an integer value.
value_of_char - Return the integer value corresponding to an English letter.
key_extender - Expands the key to make it equal in length to the plaintext.

Brute Force Attack

Following is the function to brute force the key with only ciphertexts (passed as a list) and a known key length of 4.

```
def brute_force(ciphertexts,key_length = 4):
    # Brute forces the ciphertext
    file1 = open("brute_force_output.txt", "w")
    for i in range(26):
        key = char_of_value(i)
        for j in range(26):
            key += char_of_value(j)
            for k in range(26):
                key += char_of_value(k)
                for l in range(26):
                    key += char_of_value(l)
                    plaintexts = []
                    for ciphertext in ciphertexts:
                        plaintexts.append(decrypt(ciphertext, key))
                    if check_property(plaintexts) and len(key) == key_length:
                        file1.write("Discovered Key: " + key + "\n")
                        print("Brute Force Discovered Key: " + key + "\n")
                        for p in plaintexts:
                            file1.write("Decoded Plaintext: "+p + "\n")
                        return
                    key = key[:-1]
                key = key[:-1]
            key = key[:-1]
        key = key[:-1]
```

To perform the attack, we have used **nested for loops to iterate through all the possible keys of length 4.** For each trial key, we **decrypt the ciphertexts and check if the key is valid for all** by using the function *check_property()*. If the key is correct and all the ciphertexts have been decoded correctly, we write the discovered key along with the decoded messages in a txt file named brute_force_output.txt

Check Property function

```
def check_property(plaintexts):
    # Checks if the property is satisfied
    for i in range(len(plaintexts)):
        if (plaintexts[i] == "Invalid plaintext"):
            return False
    return True
```

Since our code returns a string “Invalid plaintext” if the key used is incorrect and the message does not match the hash, this function checks for any such cases and returns a boolean value based on the same.