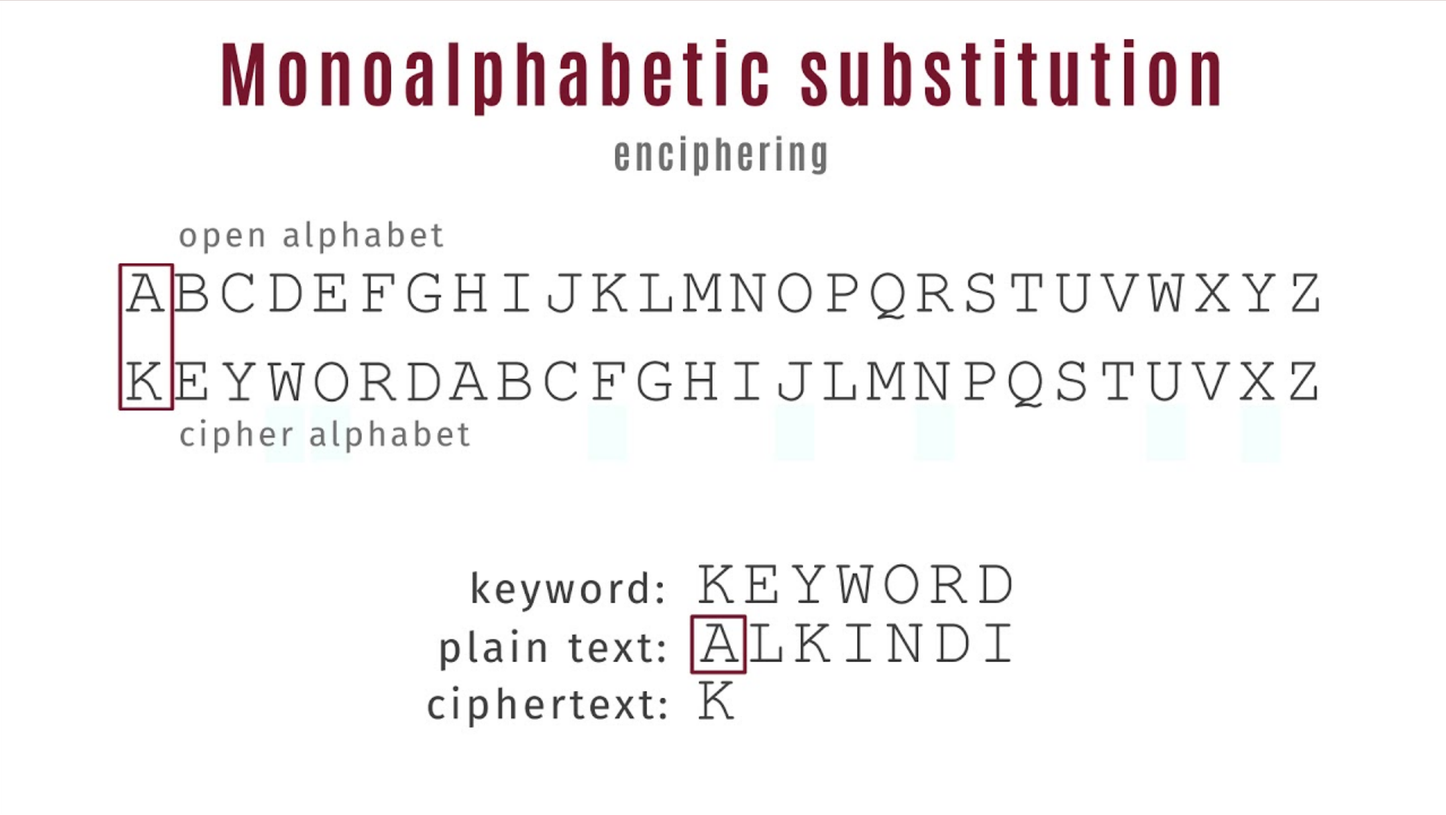
Network Security

Aditya Peer 2020355
Vedant Bothra 202060

Mono-alphabetic Subsitution Cipher

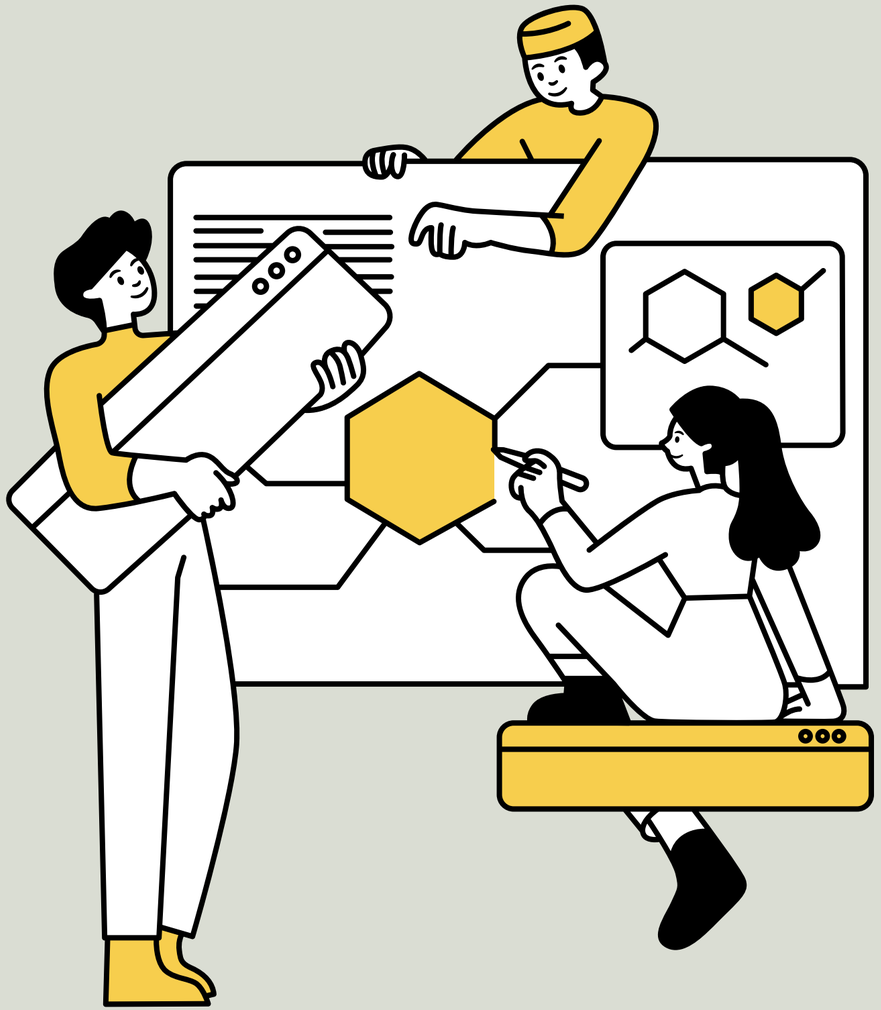


Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process.

But for this assignment instead of substituting one character , we substitute a pair and the character set is {A,B,C}.

Our Project 0

We have to implement project 0 which is mono-alphabetic substitution cipher, in which we have to encrypt and decrypt a pair of characters at a time, viz. <xy>,where x ε {A, B, C}, y ε {A, B, C}.



```python
if __name__ == "__main__":
    print ("Choices:\n1) generate a random string as user_text\n2) input pro
    choice = input().strip()
    user_text = ""
    if choice == "1":
        print("random string to be generated, enter length of the string:")
        length_s = int(input().strip())
        if length_s % 2 == 0:
            user_text = generate_random_string(length_s)
        else:
            print("length should be even, restart......")
            quit()
    elif choice == "2":
        user_text = get_plain_text()
    else:
        print("Invalid choice")
        quit()
    printt("user_text:", user_text)
    plain_text = user_text + hash(user_text,1)
    printt("plain_text:", plain_text)
    key = get_key()
    printt("key:", key)
    cipher_text = encrypt(plain_text, key)
    printt("cipher_text:", cipher_text)
    decrypted_text = decrypt(cipher_text, key)
    printt("decrypted_text:", decrypted_text)
    checkMatch(decrypted_text)
    brute_force(cipher_text, plain_text)
```

The mainFunction:

In the main function , the User has has 2 choices whether they want to enter their own plain text or the user wants to generate a random string.

Here in this example user enters choice 1 , to generate random string of even length 100.

.

```python
# function returns the hash of a given string
def hash(user_text, toPrint):
    hash_256 = hashlib.sha256(user_text.encode('utf-8')).hexdigest()
    hashstring = ""

    # MATCHING A = 0 - 5, B = 6 - 10, C = 11 - 16

    for b in hash_256:
        dec = int(b, 16)
        if dec <= 5 : hashstring += 'A'
        elif dec <= 10 : hashstring += 'B'
        else : hashstring += 'C'
```

- Here user text has been generated of length 100.
- After the user text is generated we hash it. We map the hash value in terms of 'A', 'B' and 'C'.
- We concatenate the original_string and the hash_string to get the plain text.

```
1
random string to be generated, enter length of the string:
100
user_text: ACBCBACAACAABCBCCBCBACACBAAAAAAACBCAAAABBABCBCCAABAABBBBCCBBBCCBCACCACAAACBCBAABCAAABBCBBBBBACACBAAB

##########################################################################################

hash:  AACCBAABCCAABACCCCCACCBBBACAACBCABAABBAAAACCBCAAAAABAACCBACACCAC

##########################################################################################

plain_text: ACBCBACAACAABCBCCBCBACACBAAAAAAACBCAAAABBABCBCCAABAABBBBCCBBBCCBCACCACAAACBCBAABCAAABBCBBBBBACACBAABAACCBAABCCAABACCCCCACCBBBACAACBC
ABAABBAAAACCBCAAAAABAACCBACACCAC

##########################################################################################
```

get_key()

```
# function returns a random key
def get_key():
    p = ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    e_p =  ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    # none of the key value should match check
    random.shuffle(e_p)
    # distribute
    key = {}
    for idx, p_ in enumerate(p):
        key[p_] = e_p[idx: idx+1][0]
    return key
```

After getting the plain_text, we need a key to encrypt it. So we generate a key using the get_key() function. The get_key() function returns a dictionary of key value pairs, which are used for substitution in plain_text during encryption. Here the plain text is encrypted using the key generated.

Similarly for decryption , using keys , the cipher text is decrypted.

```
##########################################################################################

key: {'AA': 'AA', 'AB': 'BB', 'AC': 'CA', 'BB': 'AB', 'BA': 'CB', 'BC': 'CC', 'CC': 'BC', 'CA': 'BA', 'CB': 'AC'}

##########################################################################################
```

Encryption and Decryption of the plain text ....

```
cipher_text: CACCCBBACAAACCCCACACCACACBAAAAAAACBAAABBCBCCCCBABBAAABABBCABCCACBABCCAAACACCCBBBBAAAABACABABCACACBBBAABCCBBBBCAACBBCBCBABCABCBBACAC
CBBAAABAAAABCCCAAAABBAABCCBBABCCA

##########################################################################################################

decrypted_text: ACBCBACAACAABCBCCBCBACACBAAAAAAACBCAAAABBABCBCCAABAABBBBCCBBBCCBCACCACAAACBCBAABCAAABBCBBBBBACACBAABAACCBAABCCAABACCCCCACCBBBACA
ACBCABAABBAAAACCBCAAAAABAACCBACACCAC
```

```python
# function takes in key and the plain text as parameters, and returns the cipher text
def encrypt(plain_text, key):
    c_text = ""
    for i in range(0, len(plain_text), 2):
        c_text += key[plain_text[i: i+2]]
    return c_text

# function takes in key and the plain text as parameters, and returns the cipher text
def decrypt(cipher_text, key):
    key = dict((values,key) for key,values in key.items())
    decrypted_text = ""
    for i in range(0, len(cipher_text), 2):
        decrypted_text += key[cipher_text[i: i+2]]
    return decrypted_text
```

In the above figure the plain text is encrypted and decrypted using the generated key.

Checking the user text received...

```
##################################################################################################################

hashing the decrypted_user_text to check if it matches the decrypted_hash
Its a match, after hashing the user_text:  AACCBAABCCAABACCCCCACCBBBACAACBCABAABBAAAACCBCAAAAABAACCBACACCAC

##################################################################################################################
```

So here is the implementation of the pi property, such that for any plain_text, we take the last 64 characters which represents the hash of the orignal_string let this be hash_orignal and rest of the plain_text represents the orignal_text. Now we hash(orignal_text) and check if it matches the hash_orignal. This is implemented in the checkMatch() function.

Brute Force to get the key...

```python
# function to perform brute force to find key
def brute_force(ciphertext):
    p = ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    e_p =  ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    dics = []
    for perm in itertools.permutations(p, len(e_p)):
        dics.append(dict(zip(perm, e_p)))
    print("total possible keys: ", len(dics))
    keysChecked = 0
    for d in dics:
        plain_text = decrypt(ciphertext, d)
        if checkMatchBruteForce(plain_text):
            print("plaintext statisfies the pi property. key found")
            print("Key Found: ", d)
            printt("total keys checked: " , keysChecked)
            break
        keysChecked += 1
    return "Not Found"
```

for each key in dics, we use them to decrypt the given ciphertext, the plain_text generated by the decrypt function is now checked for the pi property, if it satisfies the property brute force attack worked and we get the desired key which was originally used.

```
##########################################################################################

total possible keys:  362880
Key Found:  {'AA': 'AA', 'BB': 'AB', 'CB': 'AC', 'AB': 'BB', 'CA': 'BA', 'CC': 'BC', 'BC': 'CC', 'AC': 'CA', 'BA': 'CB'}
total keys checked:  14518

##########################################################################################
```

Thank You !!!