Challenge 2

Rishi Vardhan Majji
24B0969

August 2025

1 Description

Recall that a finite group is a finite set G along with a binary operation $\cdot :$ G$\times$G $\rightarrow$ G such that the binary operation $\cdot$ satisfies the following conditions:

1. (Associativity) x $\cdot$ (y $\cdot$ z) = (x $\cdot$ y) $\cdot$ z for any x,y,z ϵ G.

2. (Identity) There exists an element e ϵ G such that x $\cdot$ e = e $\cdot$ x = x for

any x ϵ G. e is called the identity element of G.

3. (Inverse) For every x ϵ G, there exists an element y ϵ G such that

x $\cdot$ y = y $\cdot$ x = e. The element y is called the inverse of x.

For instance, the set 0,1 with the binary operation of XOR forms a finite

group. The identity element is 0, the inverse of 0 is 0 and the inverse of 1 is 1. Another example of a finite group is Zn, the set of all integers modulo n for some natural number n. Given a finite group G and a positive integer m, the group Gm consisting of m-tuples of elements from G is also a group, with the group operation being co-ordinate wise application of the group operation of G.// As discussed in class, the one-time pad can actually be used for messages in any finite group. When using the group Gm, this requires that key is in the form a string of m "characters" each of which is uniformly sampled from G, independent of each other.
In this challenge, you will see a (faulty) implementation of the one-time pad over the group Zm 128 (for some m). Each message character is converted to its 7-bit ASCII representation and treated as a group element of Z128. You do not know the key used in the encryption, but you suspect that it is not uniformly random. Look at the encryption script in encryptor.py and f igure out a way to retrieve the original message.

2 Approach

Here, as per the encryptor

```
key = chr(random.randint(0,127)).encode()
    for _ in range(1, len(plaintext)):
        key += chr(hashlib.sha256(key).digest()[0]%128).encode()
```

It chooses a random integer in 0 to 127 and creates a key based on that integer. The point is that a hash is like a finger print, so for a particular integer you start with, you always get the same key at the end. So there are only 128 keys possible for this encryption, so we are going to simulate the key generating function and brute force it.

```
for i in range(0,128):
    print(i)
    print("..........")
    key=chr(i).encode()
    for a in range(1, len(encflag)):
        key += chr(hashlib.sha256(key).digest()[0]%128).encode()
```

So we are looping this for all integers from 0 to 127, creating every key.

```
    flag=b""
    for i in range(len(key)):
        flag+= chr((encflag[i]+127*key[i])%128).encode()
    print (flag)
```

And we are decrypting the cipher using all these keys , and we can just search for the one which is sensible.

```
0
..........
b'hl`w@\x1f!\t\x06JU%qB Hf=00~#:DK(1x!*)x!('
1
..........
b'g\x0fU5i\x1cvX\\fJ3Ev\x1f\x0c0,^e\x14\x7f~a%.M\x06z8\x11Ul*'
2
..........
b'f\x7fTuE[uaRn6Ux=(KK?\x00H\t\x05yYJQY\x15<\x06B|YH'
3
```

```
..........
b'eRi7\x1c\x06T>I(;\x12hU0e\x1b\n`b>W?>\x05R\x16\x02\x04\x10\x1f\x13\x19@'
4
..........
b"du_1Q\x17_O{9dO<YdB\x13&%_>\x14\x01S\x0f]mv8\x0cDi+'"
5
..........
b'cs409{algebra_enters_the_picture!}'
```

And as you can see that we got our flag for the key made from integer **5**. Hence, our key is **cs409{algebra_enters_the_picture!}**