Symmetric Encryption using private and public keys

Consider a text message "A long message". The first step is to break the message into chunks of 16 letters and they are arranged in the following fashion. (Empty cells are filled with underscores)

A	e		s
s	l	a	o
n	g	g	e
	_	m	_

Now these values are converted to corresponding ASCII values (in decimal)

65	101	32	115
115	108	97	111
110	103	103	101
32	95	109	95

Let F be the folded string i.e., the sum of ASCII values of these 16 characters. Here, **F=1492**. The table is refilled by replacing ASCII values with **F - ASCII** values.

1427	1391	1460	1377
1377	1384	1395	1381
1382	1389	1389	1391
1460	1397	1383	1397

A public key P is calculated by multiplying two randomly generated prime numbers p1 and p2.
Let p1 =
83597148959638829279794709002978352167
and p2 =
23758902996420998773367501245564160871 1

P=p1*p2 and **M=P%F**, here **M=937**

```
No of bits (n) in prime is  128

Random n-bit Prime (p):  183597148959638829279794709002978352167
Random n-bit Prime (q):  237589029964209987733675012455641608711

N=p*q= 436206685255151543855942538857045664240503864151074931288071222756797729267 37
```

The M is the **private** key.

The following Python code calculates the product of these two 128-bit prime numbers.

```
import numpy as np

# Define the two large numbers as numpy
      arrays
num1 =
      np.array([183597148959638829279794709002978352167])
num2 =
      np.array([237589029964209987733675012455641608711])

# Perform the multiplication
result = np.multiply(num1, num2)

# Convert the result back to an integer
result_integer = int(result)

# Calculate the remainder when divided by
      1492
remainder = result_integer % 1492

print("Product of the two numbers:",
      result_integer)
print("Remainder when divided by 1492:",
      remainder)
```

Output:

```
Product of the two numbers:
436206685255151543855942538857045664240503864151074931288071222756797729267 37Remainder when divided by 1492: 937
```

Now each entry **E** in the table is XORed with the private key M,
(Values are converted to binary before performing this operation and converted back to decimal)

1594	1734	1565	1736
1736	1729	1754	1740
1743	1732	1732	1734
1565	1756	1742	1756

The numbers are read from the top right corner to encrypt this table.

Each digit (0-9) is now associated with a distinct letter or special character and the number separator is represented as **T**.

```
digit_to_char = {
   '0': '%',
   '1': 'Y',
   '2': 'h',
   '3': '#',
   '4': '(',
   '5': 'x',
   '6': 'p',
   '7': ':',
   '8': '/',
   '9': 'V'
}
```

After performing the above operations, values look like this,
1594 => YxV(
1734 => Y:#(
1565 => Yxpx
1736 => Y:#p
1729 => Y:hV
1754 => Y:x(
1740 => Y:(%
1743 => Y:(#
1732 => Y:#h
1734 => Y:#(
1565 => Yxpx
1756 => Y:xp
1742 => Y:(h
1756 => Y:xp

Hence this can be read as,

TY:#pT YxpxT Y:(%T Y:#(T Y:x(T Y:#(T YxV(T Y:hVT Y:#hT Y:xpT Y:#pT Y:#hT Y:(hT Y:(#T Y:xpT YxpxT

Decrypting can be done by undoing the processes,
One of the properties of XOR that helps to decrypt is,
Let A XOR B = C, if B is unknown (A and C are known), B = A XOR C
