Group M

Dear members of the OCRAI board committee:

At our company we hope to support other companies' business endeavors through providing encryption security for sensitive data at an affordable cost. We are happy that you have reached out to us on a review from another one of our clients, and we hope that you will be pleased with the encryption method included in this letter that we have created for your company. Our mission is to enable secure communication through a standard encryption process that relies on the secrecy of the encryption key more than the actual process of encryption, so anyone with malintent will not be able to decipher the message if it is intercepted even if they know the encryption process. Using these methods, we hope to remedy the problems that you brought to us. In your letter, you described that your company has recently had security breaches through company employees accidentally sending text messages to the wrong recipients. In doing so, sensitive company information was released and could have caused serious repercussions. We understand that these messages are meant only for other company employees and need to remain confidential, even in the event that they are sent to the wrong person. These messages need to still be compatible with the mode of transmission (text messages on mobile phones) even in their encrypted state. They also need to remain a reasonable length so it does not take a significantly larger percentage of characters to send the ciphertext. And, of course, they need to be easily decrypted by other company members who are the intended recipients of the text messages through a secure and unique process. We hope our proposal of an encryption system will satisfy all of these needs to your standards.

To clearly illustrate the process of encryption and decryption, we will provide an example at each step. As stated previously, we need a key with which to encrypt and subsequently decrypt the text of the message. It can be any word with twelve or fewer characters. For example, we can choose the word, "Jerusalem" for our key. We adjust all the letters to be lowercase, and with this key, we remove any duplicate letters. Our edited key then becomes "jerusalm". We also need to divide the plaintext message up into segments of 160 characters and apply the following process to each segment. Our example will have fewer than 160 characters, but it will be illustrative enough for our purposes.

We then use the edited key to create a cyclic map of letters and apply this to our plaintext message we want to encrypt. For example, let's say we want to encrypt the phrase, "Joshua has asked me to leave my possessions in the office." We adjust all these letters to be lowercase as well, and then we use the cyclic map to write any j's as e's, any e's as r's, any r's as u's, and so on until any m's are written as j's. After this first step, our altered plaintext (which we will refer to as cipher 1) now reads, "eoahsl hla lakrd jr to mrlvr jy poaaraaiona in thr officr."

We also apply this cyclic map of letters to our alphabet, which also includes spaces (' '), periods ('.'), and commas (',') for both mathematical and security purposes. After this transformation, our alphabet "abcdefghijklmnopqrstuvwxyz .," becomes, "lbcdrfghiekmjnopquatsvwxyz .," which we will refer to as our alphabet prime.

We next create an initial shift index by adding up all of the positions of cyclically rotated letters in our zero-based alphabet. For example, the letter 'j' was originally at index 9, the letter 'e' was originally at index 4, 'r' at 17, 'u' at 20, 's' at 18, 'a' at 0, 'l' at 11, and 'm' at 12. So, our initial shift is 9 + 4 + 17 + 20 + 18 + 0 + 11 + 12 = 91. We then take 91 (mod 29) = 4 as our initial shift index. The mod 29 is because our alphabet of 26 letters has 3 additional characters appended on to it.

We then take cipher 1 and apply the following operation to it to shift the characters and call the result cipher 2: add the index of the current character in cipher 1, the initial shift index calculated in the previous step, and the position of the current character in the alphabet prime. We then take this number (mod 29) and replace the character with the resulting character at that position in our alphabet prime. For example, in our cipher 1, the first letter is ‘e’. So, we take the index of the current character 0, since we are at position 0 in cipher 1, the initial shift index 4, and the position of the current character in the alphabet prime 9, since ‘e’ is in the ninth position. We then add 0 + 4 + 9 = 13. Then the first character of cipher 2 is ‘n’ since that is the letter at the 13th position in our alphabet prime. The second letter in cipher 1 is ‘o’. So, we take the current index 1, the initial shift index 4, and the position of ‘o’ in alphabet prime 14, add them together to get 19, and get ‘t’ as the second character of cipher 2. We continue this process on until we get cipher 2 to be “ntyo,ehajcmpf.wwur sokxedlwglqldww.,pbcwllgp.rajb,wmdridgb”.

For the third and final step of the encryption, we parse the cipher 2 in such a way that we pass through and grab all the letters that are equivalent to zero (mod the length of the original key) in order and have that be the beginning of the next message, which will be referred to as cipher 3. We then pass through cipher 2 and grab all the letters that are equivalent to one (mod the length of the original key) and append it to the existing letters on cipher 3. We repeat this process up until we have grabbed all the letters from cipher 2. To best illustrate this concept, we will not only show what cipher 3 becomes, but also use colors to demonstrate the new placement of the letters. In our example, we first go through and cipher 2 and collect all the letters in the locations that are equivalent to 0 (mod 9) since “Jerusalem” has 9 characters. This would be: “nc gpri”. We then do the same for those letters that are equivalent to 1 (mod 9), which would be: “tmslbad”. Following this pattern, we would get cipher 3 to be “nc gpritmslbadypoqcjgofklwbb,.xdl,ewewlwhwdwgmaul.pdjrw,.r”. We now show the comparison of cipher 2 and cipher 3 using colors to indicate the position of the letters. Cipher 2: “ntyo,ehajcmpf.wwur_sokxedlwglqldww.,pbcwllgp.rajb,wmdridgb” becomes cipher 3: “nc_gpritmslbadypoqcjgofklwbb,.xdl,ewewlwhwdwgmaul.pdjrw,.r”.

If the original plaintext message was longer than 160 characters, we will have completed these steps for each of the blocks of text we parsed at the beginning. At this point the encryption is complete, and we send the blocks that are at most 160 characters each in the same order as the plaintext message would have been sent. Each of these steps is reversible given the key, and reversing them will yield the original plaintext message. You will want to decrypt them in blocks and not as a whole with the program.

While this encryption and decryption can be done by hand, we understand that our method can seem quite intricate. To that end, we have created a python code program that will complete the encryption and decryption for the employees. Because the key is the important part of the encryption process, the existence of the program is not a security vulnerability. We will provide the working program to you as a part of our service.

As a general overview of the process we just covered in the letter, given a plaintext message to be encoded, it will be split up into blocks, scrambled according to a key in a Caesar Cipher fashion , shifted by a scrambled alphabet and character location information in a way somewhat similar to an Affine Cipher but with slightly different technicalities, and then parsed into a pattern determined by the key. This multi-step encryption process should address the problem of confidential company information being received by unintended parties, because to any erroneous recipient, it will seem like gibberish. Regarding any party with malintent who is trying to intercept the messages, the messages will be secure as long as the malicious party does not obtain the key. While this was not stated in the original explanation of the problem in your letter to us, it is an added benefit that comes with encryption. The text messages will still be easily transmitted using mobile phones since they use the same characters as the original plaintext message being sent. The messages also remain the same length as the original plaintext, so overflow is not an issue as long as the plaintext messages are not that long themselves. Overall, we expect that this encryption will meet all of your needs and be straightforward enough to use, especially with the aid of the python program. If you have any questions whatsoever, please feel free to reach out to us. We would be more than happy to help you with any concerns.

Best regards,

[redacted]

Python Code:

```
from collections import OrderedDict


def remove_duplicate(str1):
    return "".join(OrderedDict.fromkeys(str1))

def split_text(plaintext, chunk_size):
    return [plaintext[i:i + chunk_size] for i in range(0, len(plaintext),
chunk_size)]


def encrypt(key_data, plaintext_data):
    # turn inputs into strings
    key = remove_duplicate(str(key_data).lower())
    plaintext = str(plaintext_data).lower()
    key_length = len(str(key_data))

    # create a cyclcical pattern to cycle some letters through
    cycle = key + key[0]

    # Cycle through the key letters once
    cipher1 = ""
    for char in range(0, len(plaintext)):
        location = cycle.find(plaintext[char])
        if (location != -1):
            cipher1 += cycle[location + 1]
        else:
            cipher1 += plaintext[char]

    alphabet = "abcdefghijklmnopqrstuvwxyz .,"  # a string with 29
characters, and 29 is prime
    # jumble the alphabet
    alphabet_prime = ""
    for char in range(0, len(alphabet)):
        location = cycle.find(alphabet[char])
        if (location != -1):
            alphabet_prime += cycle[location + 1]
        else:
            alphabet_prime += alphabet[char]
    cipher2 = ""

    start_position = 0
    for char in range(0, len(key)):
        start_position += alphabet_prime.find(key[char])

    for char in range(0, len(cipher1)):
        location = alphabet_prime.find(cipher1[char])
        cipher2 += alphabet_prime[(start_position + location + char) % 29]

    cipher3 = ""
    for i in range(0, key_length):
        cipher3 += cipher2[i::key_length]

    return cipher3
```

```
def decrypt(key_data, ciphertext_data):
    # turn inputs into strings
    key = remove_duplicate(str(key_data).lower())
    ciphertext = str(ciphertext_data).lower()
    key_length = len(str(key_data))

    # find out how many extra characters there are past a multiple of
key_length
    bonus_chars = len(ciphertext) % key_length
    num_iters = len(ciphertext) // key_length

    plain1 = ""
    extra_cycle = 0
    if (bonus_chars > 0):
        extra_cycle = 1
    for i in range(0, num_iters + extra_cycle):
        character_number = i
        if (i != num_iters):
            plain1 += ciphertext[i]
            for j in range(1, key_length):
                character_number += num_iters
                if (j <= bonus_chars):
                    character_number += 1
                plain1 += ciphertext[character_number]
        else:
            for j in range(0, bonus_chars):
                character_number = i + j * (num_iters + 1)
                plain1 += ciphertext[character_number]

    # create a cyclcical pattern to cycle some letters through
    cycle_prime = key + key[0]
    cycle_length = len(cycle_prime)
    # reverses the cycle for the decryption cycle
    cycle = cycle_prime[cycle_length::-1]

    alphabet = "abcdefghijklmnopqrstuvwxyz .,"  # a string with 29
characters, and 29 is prime
    # jumble the alphabet
    alphabet_prime = ""
    for char in range(0, len(alphabet)):
        location = cycle_prime.find(alphabet[char])
        if (location == -1):
            alphabet_prime += alphabet[char]
        else:
            alphabet_prime += cycle_prime[location + 1]

    plain2 = ""

    start_position = 0
    for char in range(0, len(key)):
        start_position += alphabet_prime.find(key[char])

    for char in range(0, len(ciphertext)):
        location = alphabet_prime.find(plain1[char])
        position = (location - start_position - char) % 29
        plain2 += alphabet_prime[position]

    plain3 = ""
    for char in range(0, len(plain2)):
        location = cycle.find(plain2[char])
        if (location != -1):
            plain3 += cycle[location + 1]
        else:
            plain3 += plain2[char]

    return plain3



encryption = encrypt("key goes here", "plaintext goes here")
print(encryption)

decryption = decrypt("key goes here", "ciphertext goes here")
print(decryption)
```