Group F

OCRA Creative Recursive Acronyms, Inc.
485 Primality Way
Provo, UT 84604

Dear OCRAI,

The use of text messaging for sending sensitive information can be dangerous as mistakes can be made and attackers will take advantage of this vulnerability. To protect your company's trade secrets, we have developed a cipher with which to encrypt and decrypt your messages. Solving this problem requires taking into account the drawbacks of using the system of a cell phone, which limits the maximum key size and message length used in our cipher.

To successfully keep your messages secure with these limitations, we considered multiple methods an attacker might use and developed the appropriate precautions. A common method of cryptanalysis includes character frequency analysis. To throw off attackers, we chose to use a more randomized method of encrypting characters than a simple substitution cipher. We implemented a Chain Code so that each character undergoes a unique transformation based on both its position in the plaintext and value, rather than simply its value.

However, this is not enough to ensure security for your company's messaging system. Even with our coded text, attackers gain valuable information from the ordering of characters in the message. For example, using a chosen-plaintext attack, eavesdroppers may deduce the key simply by comparing the found plaintext and its associated ciphertext. Therefore, we created a two-stage cipher that utilizes keyed columnar transposition to further randomize the encrypted message and throw attackers off the ordering of the message. This will make the key more useful to the encrypting of the plaintext.

Below is a description of the cipher. We believe that, despite the described limitations, our solution will significantly increase the security of your messaging system and protect your company's sensitive data.

Cipher Overview

Our cipher is constructed from a Chain Code and a keyed columnar transposition. Our cipher takes in a key of 10 unique characters, a plaintext, and produces a ciphertext. At a high level, encryption happens by converting the plaintext into integers (mod 26) using the A0-Z25 mapping. The plaintext is then padded with 'X' characters to a length that is a multiple of 10. The plaintext is then added to the keystream (mod 26) generated by the Chain Code. This result is finally put through a keyed columnar transposition, and the integers are converted back to ciphertext characters using the same A0-Z25 mapping. Decryption follows this same process, just backwards.

Chain Code:

This component takes inspiration from linear-feedback shift register (LFSR) based keystream generators, and functions as follows. Using the ten character key, converted to integers by A0-Z25 mapping, as an initial state (k_0,...,k_9), additional keystream characters are generated using the linear recurrence relation…

$$k_{i+10} = k_i + k_{i+1} + k_{i+4} + k_{i+6} + k_{i+9} \pmod{26}$$

This recurrence relation is used to generate a keystream of length equal to the length of the plaintext. This relation bears some resemblance to a LFSR with taps in the 1, 2, 5, 7, and 10 positions. These taps are chosen to increase the periodicity of the keystream.

To encrypt using the Chain Code, simply add the plaintext character to the keystream character mod 26. In other words…

$$c_i = p_i + k_i \pmod{26}$$

This ensures that each plaintext character is mapped to a ciphertext character in a simple, but dynamic way.

To decrypt using the Chain Code, simply add the ciphertext character to twenty-six minus the keystream character mod 26. In other words…

$$p_i = c_i + (26 - k_i) \pmod{26}$$

This decryption works because we are simply adding the additive inverse of the keystream character k_i to each ciphertext character c_i, thus canceling the effect of the key.

Keyed Columnar Transposition:

This component is included as a defense against keystream reconstruction attacks on the Chain Code using predicted or known plaintexts. This transpose works by using the 10 key characters as the key for a simple columnar transposition.

In the keyed columnar transposition, the message is written out in rows of a fixed length, and then read out again column by column, and this column order is chosen depending on the key. For example, the key ZEBRAS is of length 6 (so the rows would be of length 6), and the permutation is defined by the alphabetical order of the letters in the key. In this case, the order would be "6 3 2 4 1 5". Suppose we use the key ZEBRAS and the message WE ARE DISCOVERED. FLEE AT ONCE. We can write this into the grid as follows:

6 3 2 4 1 5
W E A R E D
I S C O V E
R E D F L E
E A T O N C
E Q K J E U

The encrypted message would then be read out as EVLNA CDTES EAROF ODEEC WIREE. To decrypt this, the receiver has to work out the column lengths by dividing the message length by the key length. Then they can write the message out in columns again, then reorder the columns by reforming the key word.

Overall, this cipher is an effective solution due to the ease of implementation, long keystream periods, and Irwin–Hall uniform output distribution. The Chain Code component effectively thwarts any sort of character frequency analysis, due to the effects of the Von Neumann extraction procedure and an additive distribution transformation. Due to simplicity concerns, irregular stepping/clocking, non-linear tapping, and non-linear key state mixing techniques were not included in the Chain Code design. Alone, this means that the Chain Code component is vulnerable to linear cryptanalysis using either a chosen or predicted plaintext, and can be broken using the modified Berlekamp-Massey algorithm. To counter this, we added stage, the keyed columnar transposition, which restricts the ability of an attacker to perform such an attack. This cipher is not without vulnerability, but it offers a simple design and provides modest security given the design restrictions.

Best regards,

Cipher Design Team
IMC

APPENDIX A: Python3 Cipher Demo Implementation

```
import re # for regex
import math # for math purposes

def strip_text(text):
    return re.sub('[^A-Za-z]+', '', text)

def pad_text(text,block_size):
    while((len(text) % block_size) != 0):
        text += "X"
    return text

def string_to_num_array(input):
    input = input.upper()
    result = []
    for x in range(len(input)):
        char = ord(input[x]) - 65
        result.append(char)
    return result

def num_array_to_string(input):
    result = ""
    for x in range(len(input)):
        char = chr(int(input[x] % 26) + 65)
        result += char
    return result

def generate_chain_encryption_key(key, length):
    i = 0
    while (len(key) < length):
        key.append((key[i] + key[i+1] + key[i+4] + key[i+6] +
key[i+9]) % 26)
        i += 1
    return key

def generate_chain_decryption_key(key, length):
    key = generate_chain_encryption_key(key, length)
    for i in range(len(key)):
        key[i] = 26 - key[i]
    return key

def columnar_encrypt(msg, key):
    cipher = ""
    k_indx = 0
    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))
    col = len(key)
    row = int(math.ceil(msg_len / col))
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)
    matrix = [msg_lst[i: i + col]
              for i in range(0, len(msg_lst), col)]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        cipher += ''.join([row[curr_idx]
                           for row in matrix])
        k_indx += 1
    return cipher

def columnar_decrypt(cipher, key):
    msg = ""
    k_indx = 0
    msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)
    col = len(key)
    row = int(math.ceil(msg_len / col))
    key_lst = sorted(list(key))
    dec_cipher = []
    for _ in range(row):
        dec_cipher += [[None] * col]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        for j in range(row):
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
            msg_indx += 1
        k_indx += 1
    try:
        msg = ''.join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot handle repeating letters
in key.")
    null_count = msg.count('_')
    if null_count > 0:
        return msg[: -null_count]
    return msg

def encrypt(plaintext, key):
    plaintext = strip_text(plaintext)
    plaintext = pad_text(plaintext, 10)
    key_array = string_to_num_array(key)
    plaintext_array = string_to_num_array(plaintext)
    chain_code_encryption_key =
generate_chain_encryption_key(key_array, len(plaintext_array))
    ciphertext_array = []
    for i in range(len(plaintext_array)):
        ciphertext_array.append((plaintext_array[i] +
chain_code_encryption_key[i]) % 26)
    ciphertext = num_array_to_string(ciphertext_array)
    ciphertext = columnar_encrypt(ciphertext,key)
    return ciphertext

def decrypt(ciphertext, key):
    key_array = string_to_num_array(key)
    ciphertext = columnar_decrypt(ciphertext,key)
    ciphertext_array = string_to_num_array(ciphertext)
    chain_code_decryption_key =
generate_chain_decryption_key(key_array, len(ciphertext_array))
    plaintext_array = []
    for i in range(len(ciphertext_array)):
        plaintext_array.append((ciphertext_array[i] +
chain_code_decryption_key[i]) % 26)
    plaintext = num_array_to_string(plaintext_array)
    return plaintext

##### testing values  below #####
key = "YELZOWSUBA"
encrypted = encrypt("According to all known laws of aviation, there
is no way a bee should be able to fly. Its wings are too small to get its
fat little body off the ground.", key)
decrypted = decrypt(encrypted,key)
print(encrypted,decrypted)
```