ICSI 526 - Spring 2023 - Homework 1

Jacob Clouse

February 16, 2023

1 - Question 1a Answer:

First of all, this IS breakable. Here is my explanation of why:

For my First name: J A C O B, I found that the combine total is 9 + 0 + 2 + 14 + 1. (9 + 0 + 2 + 14 + 1) is equal to 26, so we use (26) mod 26 which is equal to 0. So in C1 = (a * P1 + b) mod 26, C1 is equal to 0 (for the most common letter E). E is normally valued at 4 in plaintext.

For my Last name: C L O U S E, I found that the combine total is 2 + 11 + 14 + 20 + 18 + 4. (2 + 11 + 14 + 20 + 18 + 4) is equal to 69, so we use (69) mod 26 which is equal to 17. So in C2 = (a * P2 + b) mod 26, C2 is equal to 17 (for the second most common letter T). T is normally valued at 19 in plaintext.

Here are the equations:
For E / First name: **C1 = (a * P1 + b) mod 26** *OR* **0 = (a * 4 + b) mod 26**
For T / Last name: **C2 = (a * P2 + b) mod 26** *OR* **17 = (a * 19 + b) mod 26**

To find the difference between the two we can subtract the first from the second:

$$17 = (a * 19 + b) mod 26 \tag{1}$$
$$0 = (a * 4 + b) mod 26 \tag{2}$$

Subtracting (2) from (1) yields:

$$\mathbf{17 = 15a \bmod 26} \tag{3}$$

We now need to take this function and solve for a. To do this, we need to move the mod operator over in (3) to the left hand side. We now have:

$$17 mod 26 = 15a \tag{4}$$

We use the Euclidean Algorithm to find the Greatest Common Divisor (or GCD) of 15 and 26 and check to see if its equal to 1. It turns out that the GCD between 15 and 26 is 1.

So this becomes:

$$17 * 15^{-1} mod 26 = 1 \tag{5}$$

Then:

$$17 * 7 mod 26 = a \tag{6}$$
$$119 mod 26 = a \tag{7}$$

Finally, we find that:

$$\mathbf{a = 15} \tag{8}$$

Now we need to solve for b. We do this by substituting in our a value for one of our equations:

$$17 = (\mathbf{15} * 19 + b) mod 26 \tag{9}$$

Then:

$$17 = 285 + bmod26 \tag{10}$$

$$(17 - 285)mod26 = b \tag{11}$$

$$-268mod26 \tag{12}$$

Finally, we find that:

$$\mathbf{b = 18} \tag{13}$$

To check our work we need to substitute b into the equation and solve it:

$$(15(19) + 18)mod26 \tag{14}$$

This is equal to 17, which is the value we calculated previously. So it works!

2 - Question 1b Answer:

The answer to 1b depends on if the **d** in **C = [a × (P-d) + b] mod 26** a constant or part of the key.

- i) If this just a constant being added in, we **CAN** crack this! It basically, it would be similar to the offset that is already being conducted on the on the plaintext. We could use the two equations to mathematically solve for it like we did with a and b.
- ii) If this is part of the key, we **CAN NOT** crack this. If it was something like a One Time pad, the encryption key is a random number and, the key is used only once. That would mean we couldn't solve for it like we did for a and b, it would have no correlation between the two equations.

3 - Question 2 Answer (Design Overview):

The design of my algorithm can be broken down into three parts:

- Substitution Section
- Transposition Section
- One Time Pad Section

I originally had coded each piece separately, tested them individually and then combined them into one overarching program (the final Product Cipher). The first thing I do inside of my Product Cipher is to get the plaintext that the user wants to encrypt. Once they enter it in, we move onto the **substitution section**. The substitution section is a spin on the classic Caesar/Affine cipher we have gone over in class. We map values for all the letters of the alphabet to number values so we can then do a displacement. We translate our plaintext into number form and ask the user to give us the displacement amount. We then combine the plaintext value and the offset together, then modulus it by 26. We return this value out of the function and feed it into the **transposition section**).
The transposition section is loosely based off of the row transposition cipher. The user gives us input, that input is then stored into a 2D matrix and then they choose a key for this matrix. For simplicity's sake, I limited my cipher to have 4 columns in total. This would potentially cut down on the security of the algorithm, but combining this section with the other two more than makes up for it. We receive the substitution ciphertext directly from the code and then move it into our 2D matrix. The length of the plaintext has to be a multiple of 4, so we have to add padding if it is not. We then query the user to give us the column order key by providing us with a combination of 1,2,3 and 4. This is then used to reorganize the matrix and transpose it back into a string, giving us our transposition ciphertext. The only thing left to do is the **one time pad section**.
What makes this cipher truly secure is the one time pad section. The one time pad as a concept was created over 100 years ago and it is still being used in cryptography to this very day because it is so secure. We use our transposition ciphertext as the input to our one time pad function and it determines the number its length. It then goes and generates a random value between 0 and 9 for each character. This gets us a string of numbers the same length of our key. We then combine this key with our plaintext using ord() operations. This should only be breakable if you have the specific key that goes along with your ciphertext. Otherwise, this ensures the security of the full cipher. Our cipher and key are stored as pickle objects and printed out in our terminal. Separate the key from the cipher, and it should be pretty dang secure.

To decrypt our cipher, all we need to do is reverse our process. First, we start off with the one time pad section, you just need to provide the pickle files for the key and the ciphertext. It will open up the pickle files and reverse the operation (done the same way you encrypted them). This is followed by the transposition section. The output from the one time pad decrypt is fed into the transposition decrypt function, all it needs from you is the column combination key that you set when you encrypted it. Once you provide that, it will rearrange the columns back to their decrypted state. But we are not done, we just have to do the substitution cipher decrypt. The output from the transposition decrypt is fed once again into the substitution cipher decrypt function, all we need to do is provide it the offset that you originally set during the encryption. And, just like that, we are done with the decryption!

4 - Question 2a Answer:

I would wager that this algorithm would be very hard to break with just pen and paper. Even if this was just the substitution and the transposition section, it would not be fun to crack. Combining two ciphers together helps to defuse the output, making it harder for a potential attacker to brute force the output. You can use frequency analysis to see which letters occur most often (since a letter will not have multiple outputs), so the first part can be cracked.
BUT where this cipher truly begins to shine is in the last section: the one time pad section. This section enables letters to lead multiple potential outputs. And, as long as you use truly random keys, no two combinations will lead to exactly the same output (we can try encrypting the same word twice and we will get two different outputs). Unless you know the key, you shouldn't be able break this with pen and paper.
Lets look at an example to try and analyze it by encrypting my first and last names: **jacobclouse**. We would start off with the substitution portion, lets use an offset key of 4. We get an output of "**negsfgpsywi**" as the encrypted output, this would be simple to break as each letter has only one corresponding output (ex: both 'o's in my name correspond to 's'). If we used a known plaintext attack, this section isn't going to stand up well. Then we move on to the transposition section, lets choose an order of 4,3,2,1. This gives us an output of "**ss gpiegwnfy**", so it certainly doesn't look like my name anymore. Normally, if we just did the transposition cipher by itself, the output would be much easier to reverse engineer with a plaintext attack (especially if there were only a few columns in the matrix). However, since we first did a substitution encryption before doing the transposition, it becomes MUCH harder to see what the original message inside it. And yet, this is still crack-able, even if it is not so easily done. Next we have the One time pad, and this doesn't take any input from the user. It gets the length of my name in cipher text form (which is 12 characters), and it generates a 12 character length string of random numbers (in this case it was '**145662785460**'). It then use the XOR operation to combine this with my previous cipher text, making the array: "**['B', 'G', 'NAK', 'Q', 'F', '[', 'R', '_', 'B', 'Z', 'P', 'I']**". The only way you can get this is by using the original, randomly generated key with it. Now, computers do generate PSEUDO-random numbers, so you could theoretically attack the random number generator within the computer system itself, but that would be a huge undertaking.

5 - Question 2b Answer:

Two Attacks that my cipher is secure against:

- **Ciphertext-Only Analysis:** As stated before, you can't break this cipher with frequency/statistical analysis because of the one time pad. One letter can have multiple outputs and multiple letters can share the same output. It is very hard to break unless you have the key.
- **Chosen-Plaintext Analysis:** Based off of the different combinations of substitution, transposition and one time pad sections, it would be near impossible to reverse engineer the key. Just because of the one time pad alone with its one off random key, you should only be able to get decipher this if you already have the key. Add to that the complexity that the substitution and transposition ciphers add on top and it become a momentous undertaking. Plaintext messages will generate with entirely different outputs depending on the combination of substitution offset, transposition order, and the random number generated within the one time pad.