Transposition, Padding, and Product Cipher Lab

Overview

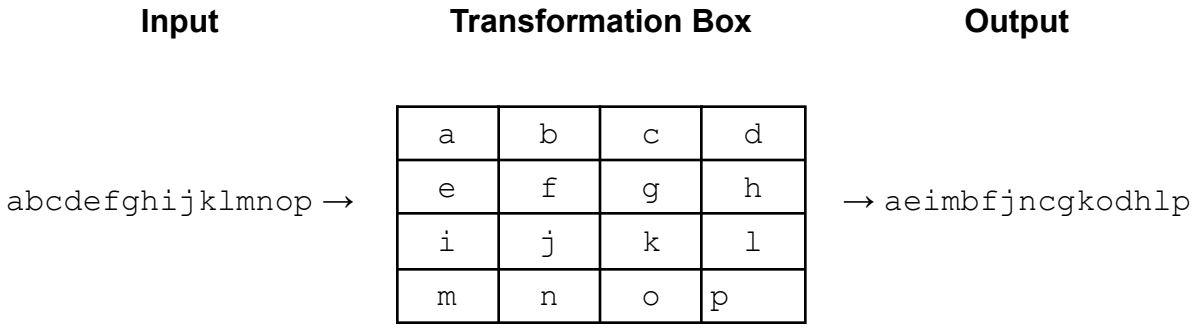
In this exercise, you and a partner will experiment with a Columnar Transposition Cipher (CT). This will give you hands-on experience working with transposition ciphers and coping with aspects of block ciphers, such as padding.

This is a **terrible** encryption algorithm! Do not ever use it for anything important! Like, not *ever*!

Part 1: Columnar Transposition

A Columnar Transposition (CT) works by using a grid to “shuffle” the letters in a plaintext. You start by writing symbols into the top-left of a grid of a particular size (which we call a transposition box), moving to a lower row when an upper row is full. Once the box itself is full, it is read out in columns top to bottom, left to right (hence the name).

For example, given the input “abcdefghijklmnop” and a 4x4 grid, a CT cipher would perform the following process:



To decrypt, you simply perform the same process again, but this time starting with the ciphertext and obtaining the plaintext as output.

CT ciphers are distinct from the Caesar and Vigenère ciphers in at least two important ways. First, CT ciphers perform *transposition*, unlike Caesar and Vigenère which only perform *substitution*. Secondly, Caesar and Vigenere are *stream* ciphers, encrypting one symbol at a time, while CT is a **block cipher**. A block cipher cannot perform encryption until and unless a certain amount of data (the block size) is available. This means that CT ciphertext will be written out in discrete blocks. This will require padding the input so that its size is a multiple of the blocksize, which we’ll discuss in a little bit.

Does the CT use a key? The block size for the CT is the square of its dimension. So, if the dimension is 4, the block size is 4 * 4 = 16. Most real-world ciphers (e.g., AES) have a fixed block size – for example, AES always uses 128-bit blocks (16 bytes). However, CT works with *any* size dimension the sender and receiver agree upon. In this way, the “key” for the CT is the dimension of the grid – not a very strong key, since it has to be an integer, and very large values are not very useful (why)?

Tasks

1. Using a 2x2 grid, encrypt a 16-character message of your choosing. Pass it to your partner. (You’ll need four blocks.)
2. Decrypt your partner’s encryption and make sure it is correct.
3. Using a 4x4 grid, encrypt a different 16-character message of your choosing. Pass it to your partner. (You’ll only need one block.)
4. Decrypt your partner’s encryption and make sure it is correct.
5. Using a 1x1 grid, encrypt the message “cafe”. Pass it to your partner. (You’ll need four blocks.)
6. Decrypt your partner’s encryption and make sure it is correct.
7. Answer the questions on Canvas.

Part 2: Padding for CT

Block ciphers encrypt and decrypt a fixed number of bytes at once, so we can't run the algorithm unless there is enough input to fill a whole block. As a result, if the message ends before the block is full, a block cipher must pad the input *before* encrypting it. However, this padding must be done in a way that can be unambiguously removed upon decryption. In other words, the original length of the plaintext must be able to be recovered, even if the plaintext looks random or even looks like padding!

Suppose d is your CT grid dimension, so the block size is d^2. Such a CT cipher must have d^2 bytes (symbols) before encrypting or decrypting. If there are not d^2 bytes of input available, you should pad the remaining space in the buffer by inserting an 'X' in the first open spot (serving as a flag) followed by as many 'Y' bytes required to fill the block being padded. Usually, this means that the padding consists of an 'X' and some number of 'Y's. However, if there is only one empty byte in the block, you would pad with just 'X'.

It turns out that padding is also used to indicate the end of the plaintext, so, if the plaintext fills the last block completely, we will create a complete additional "padding block" consisting of 'X' followed by enough 'Y' bytes to fill the block.

Thus, there are **three** cases for padding, and once we reach the end of the input, one case will *always* apply:

1. There is only space for one symbol. Add an 'X'.
2. There is space for >1 symbols. Add an 'X' in the next spot and fill the remaining space with 'Y's.
3. The plaintext ends exactly at the end of a block. Fill an entire block with padding.

When decrypting, remove the padding and ignore it.

This means that you'll always have to add padding. Either you'll add it to the last block of input, or you'll add a full padding block after the input ends.

In case you're interested, this particular "XYYY…" approach to padding was suggested by Schneier and Ferguson, but there are many block cipher padding schemes.

Padding Examples

In these examples, assume a four-byte (2x2) block.

1. Final block: "00". Pad the last plaintext block before encryption so that it reads "00XY". After decryption, "00XY" will be the last block. Remove any trailing Ys and the final X so that it again reads: "00"

2. Final block: “X”. Pad the last plaintext block before encryption so that it reads “XXYY”. After decryption, remove the trailing Ys and the final X so that it reads “X”. (See how the algorithm keeps the actual X in the message?)
3. Final block: “ABC”. Add an X in the last spot so that it reads “ABCX”. On decryption, remove any trailing Ys (there are none!) and then remove the final X, resulting in “ABC”.
4. Final block: “ABCD”. There is no room for padding, so add a full padding block “XYYY” to the plaintext before encryption. On decryption, the second to the last block will decrypt “ABCD”. The final block will decrypt “XYYY” and should be ignored after decryption.

Tasks

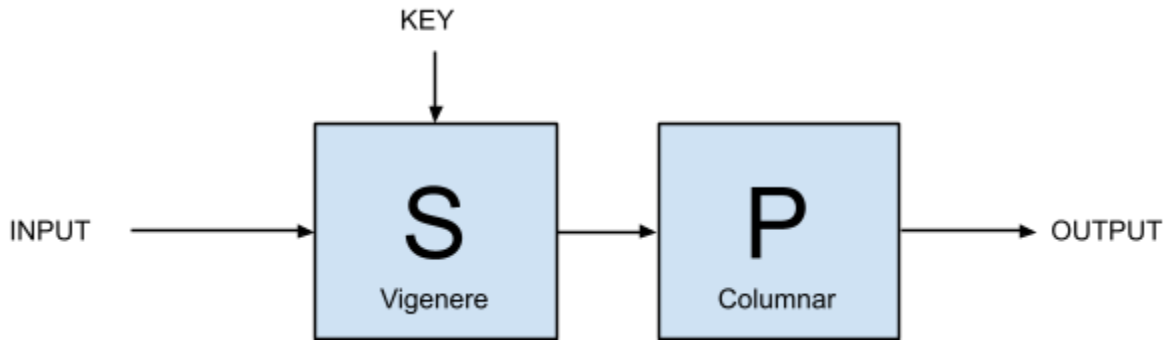
1. Using a block size of 4 (e.g., a 2x2 grid), write the message “A” into the grid and pad it as required. Pass it to your partner.
2. Unpad the message from your partner, ensuring that they padded it correctly.
3. Using a block size of 4 (e.g., a 2x2 grid), write the message “XYY” into the grid and pad it as required. Pass it to your partner.
4. Unpad the message from your partner, ensuring that they padded it correctly.
5. Using a block size of 4 (e.g., a 2x2 grid), write the message “1234” into the grid and pad it as required. Pass it to your partner. (Hint: you’ll need two blocks!)
6. Unpad the message from your partner, ensuring that they padded it correctly.
7. Using a block size of 16 (e.g., 4x4 grid), write the message “A” into the grid and pad it as required. Pass it to your partner.
8. Unpad the message from your partner, ensuring that they padded it correctly.
9. Answer the questions on Canvas.

Part 3: Product Ciphers

A Product Cipher is a cipher that uses multiple transformations in series. For example, a Vigenère Cipher (VC) and a Columnar Transposition (CT) can be combined as follows:

$$E_{Columnar}(\text{dim}, E_{Vigenere}(key, plaintext)) = ciphertext$$

This example first performs Vigenere substitution using 'key' on 'plaintext' and then performs a Columnar Transposition with a *dim * dim* block on the resulting output. Cryptographers call substitution functions *S-boxes* and transposition functions *P-boxes* (P for permutation). So, this product cipher performs one S-box and one P-box, like so:



Decryption can be performed using the individual decryption algorithms *in reverse.* Note that you must do the inside operations first. So, first reverse the CT, and then undo the VC:

$$D_{Vigenere}(key, D_{columnar}(\text{dim}, ciphertext)) = plaintext$$

Multi-Round Product Ciphers

Product Ciphers often make use of multiple **rounds** where the same steps are performed some number of times. For example, if we collapse the previous encryption functions together as follows:

$$E_{Columnar}(\text{dim}, E_{Vigenere}(key, plaintext)) \rightarrow E_{product}(\text{dim, key, plaintext})$$

we can express **two rounds** as:

$$E_{product}(\text{dim, key, } E_{product}(\text{dim, key, plaintext})) = \text{ciphertext}$$

we could further “collapse” this function like so:

$$E_{product}(\text{dim, key, num\_rounds, plaintext}) = \text{ciphertext}$$

Decryption would use the appropriate functions in reverse. Collapsing the decryption functions:

$$D_{Vigenere}(\text{dim}, D_{Columnar}(key, \text{ciphertext})) \rightarrow D_{product}(\text{dim, key, ciphertext})$$

we can express two rounds as:

$$D_{product}(\text{dim, key, } D_{product}(\text{dim, key, ciphertext})) = \text{plaintext}$$

… and we can further collapse this function too:

$$D_{product}(\text{dim, key, num\_rounds, ciphertext}) = \text{plaintext}$$

For this part of the lab, you will implement the simple Product Cipher just described. In other words, you will first perform VC with the given key and then perform CT using the dimension 4 (a block of 16 bytes) for a given number of rounds. You will implement both encoding and decoding mechanisms.

Product Ciphers: Do Padding Once, and do it *First*

Padding is still required for this cipher, since the product cipher is a block cipher.

However, it is not obvious how to combine the stream cipher of Vigenère with the block cipher of the Columnar Transposition. One approach might be to think of “one round” of our Product Cipher as:

1. Vignère be performed on the whole file
2. Columnar on the file (block by block), padding the last block

There's a problem with this approach. When round two occurs, Columnar will again add padding (since CT always adds padding to the end of the input). Each of *n* rounds will add more padding to the end of the file, but the padding blocks will not all have the same number of rounds (the last block of padding will not have gone through Vigenère substitution, for example).

To illustrate why this is not a good approach, suppose we are using a two round cipher as described. We have the input “ABCDE”, we are using a dimension of 2 (block size of 4), and our Vigenère key is “1111” (we are going to simply shift each letter by 1 position in our heads). If we do VC on the whole input first, we get:

Round 1

```
VC(11111, ABCDE) → BCDEF
```

Then we would do a CT:

```
BC
DE → BDCE

FX
YY → FYXY
```

… giving us a Round 1 output of:

```
BDCEFYXY
```

Round 2

Picking up with our Round 1 output:

```
VC(11111111, BDCEFYXY) →
CEDFGZYZ
```

Then, another CT:

```
CE
DF → CDEF

GZ
YZ → GYZZ
```

And more padding (since we ended on a block division!)

```
XY
```

```
YY → XYYY
```

… giving us a final output of: `CDEFGYZZXYYY`, which is much larger than our input and, as promised, the last padding block is unciphered. Boo.

Instead, the entire cipher should **function** as a block cipher, encrypting each block of input, first using Vigenère and then using Columnar. Each set of *n* rounds is performed **on a single block** before moving to the next block, enabling you to reuse the same structures. When the **very last** block of the input is reached, that plaintext block is padded to the block size *before* Vigenère and Columnar are applied in the first round. Then in the second (and future) rounds, the block is already full and should not be padded again. On decryption, the transformation rounds are applied, and, if it is the last block in the file, the padding is removed before the plaintext is written out.

Using the same example conditions as above:

Block 1, **Round 1**

```
VC(1111, ABCD) → BCDE
```

Then we would do a CT:

```
BC
DE → BDCE
```

Block 1, **Round 2**

```
VC(1111, BDCE) → CEDF
```

Then we would do a CT:

```
CE
DF → CDEF
```

Then, we would move on to our **next block...**

Block 2, Round 1

We only have 'E' as our input, so **we need to pad the block**. Thus, 'E' becomes:

```
EXYY
```

Now we can do our rounds:

```
VC(1111, EXYY) → FYZZ
```

Then we would do a CT:

```
FY
ZZ → FZYZ
```

Block 1, Round 2

Now on to round two. Our block is already full (since we padded it) so **we don't need to** do any **more padding.**

```
VC(1111, FZYZ) → GAZA
```

Then we would do a CT:

```
GA
ZA → GZAA
```

… giving us a final output of: `CEDFGZAA`. This output is much shorter and leaks much less information.

To summarize: Read in blocksize bytes of input. If you have a full block, perform Vigenère and Columnar in sequence for *n* rounds and write it to disk. Then, read in the next blocksize bytes and repeat the process until you run out of input. Pad only the last block, and pad it before doing any encryption.

To decrypt, reverse the process. Note however that to remove the padding from the last block, you'll need to do Vigenère and Columnar for *n* rounds *first.* You'll remove the padding as the last step before writing the output.

Task

1. Using the product cipher described here, encrypt the message "A" using the padding scheme described above, the Vigenere key "ABCD" (shift values 0, 1, 2, 3), and a 2x2 CT grid. You should get the ciphertext "AAYB".
2. Decrypt the message "AAYB" using the product cipher decryption described above.
3. Multi-round product cipher. Use the product cipher described here, except this time go through the Vigenere and Columnar steps *twice.* Encrypt the message "ABCD" using the padding scheme described above, the Vigenere key "ABCD" (shift values 0, 1, 2, 3), and a 2x2 CT grid. Since the input ends on a block boundary, the steps will be as follows:

a. First block:
    i. Encrypt the first block (“ABCD”) using Vigenere – the block is full so it doesn’t need padding.
    ii. Encrypt the previous result using CT and a 2x2 grid.
    iii. Encrypt the previous result using Vigenere and the key.
    iv. Encrypt the previous result using CT and a 2x2 grid.
b. Second block:
    i. The block is empty, so pad the full block.
    ii. Go through steps i-iv with the padding block as for the first block.
c. Compare your result with your partner. You should have two blocks: “AEFJ XBBE”

4. Decrypt the previous result and ensure you get “ABCD” as the plaintext.
5. Answer the questions on Canvas.

Extra Credit

There are two extra credit opportunities for this assignment.

EASY: For 1% on top of your earned points, **implement a product cipher** in your favorite language. It must be your own work and should not contain any pieces of code from online sources, or be AI-generated. For more details of the extra credit option, please contact TA / instructor.

HARD: For 5% extra credit, implement a multi-round product cipher for encryption and decryption. Use the encryption and padding scheme described in the lab manual.

For full credit, your implementation must meet the following requirements:

1. The code should read the Vigenere key and input text from a file and write the output into another file.
2. Input and key should not be limited to alphanumeric only. In other words, your code should work with any binary input as plaintext or key.
3. Mode (encode/decode), vigenere key path file, input path file, output path file, columnar dimension, and round number should be read as command line parameters. For example:

   ```
   product.py encode /users/home/key.txt /home/input.txt /home/encode.txt 4 2
   ```

   …should encrypt the input.txt file with key.txt, and a columnar matrix of 4X4, and it will run for two rounds.
4. If you use any other language than C, C++, Java or Python, please write thorough documentation so that someone who does not know the language can have a basic understanding of your code.

If your code meets all the requirements, you'll get full extra credit which will be added to your grade at the end of the semester. If not all requirements are followed, you might not get all 5%.

Submitting your Extra Credit Project

Push your code into https://github.umn.edu/ and add TA and instructor as a collaborator. If you don't know how to use git, please follow the instructions in this document:
When you push your code and it is ready to be graded, please email your TA.