Elementary Encraption 2

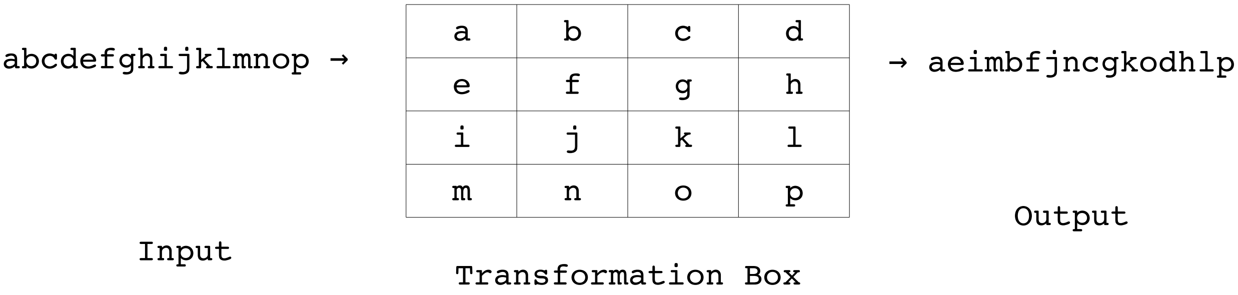
Implementing a Columnar Transposition in C

Dr. Peter A. H. Peterson <pahp@d.umn.edu>
University of Minnesota Duluth

Overview

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

A Columnar Transposition (CT) works by writing symbols into 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:



a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

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, CT ciphers are block ciphers in that the block cannot be written out until the P-box (permutation box) is full or sufficiently padded. The latter means that, while Caesar and Vigenère ciphertext can be written letter by letter (stream cipher), CT ciphertext will be written out in discrete blocks (block cipher). This will require padding the input so that its size is a multiple of the blocksize. (See Requirements, below.)

Requirements

Code and Compilation

1. Your submission will be electronically graded for correctness and uniqueness. Therefore, your code must successfully compile on the MWAH 187 machines running Ubuntu.
2. Your code **must not** use any of the Banned Functions listed in the Vigenère Encraption lab manual or it will be scored a **zero (0)**.
3. You need to create a single C source file capable of generating object files for both an encoder and decoder. Your code will be compiled using the commands

   ```
   gcc columnar.c -o encoder -D MODE=ENCODE
   ```
   and
   ```
   gcc columnar.c -o decoder -D MODE=DECODE
   ```
   See the Appendix of the Vigenère lab manual for instructions to do this.
4. Your code should not use any libraries explicitly intended for encryption. In particular, you

must NOT use other online examples of the Columnar Transposition as a coding template; if you do not understand how the TC works, talk to your instructor or TA.

5. Instead of keys, the dimension of the transposition box will be specified on the command line.
6. The transposition box dimension will be between 1 and 10 .
7. Input will not be limited to text only (i.e., test using binary data like applications or images).
8. Input will not be limited to any particular length (i.e., test using large files).

Encoder and Decoder

Your encoder and decoder **must** take the following parameters as input:

1. The number of rows and columns in the transposition box, e.g., 4. (The box will be a square)
2. A path to an input file, e.g., “/tmp/input”
3. A path to an output file, e.g., “/tmp/output”
4. Upon running 'encoder 4 /tmp/input /tmp/output', output should contain the contents of input as encoded using a 4x4 transposition box. Upon running 'decoder 4 /tmp/encoded /tmp/decoded', decoded should contain the contents of encoded as decoded using a 4x4 transposition box.

Padding

Block Ciphers encrypt and decrypt a fixed number of bytes at once. If there is not enough input to fill whole blocks, a block cipher must pad the input before encrypting it. However, this padding must be done in a way that can be undone upon decryption. In other words, the original length of the file must be able to be recovered.

If *x* is the number of columns and *y* is the number of rows in your transposition box, your CT cipher should read in *xy* bytes of input at once and then encrypt it. If there are not *xy* bytes available, you should pad the remaining space in the buffer with the byte 'X' (serving as a flag) followed by as many '0' bytes required to fill the block. If the plaintext fills the last block completely, your cipher should create a “padding block” consisting of 'X' followed by enough '0' bytes to fill the block. To be completely explicit, by 'X' we mean the character 'X' and by '0' we mean the character zero, not the number zero or the character “capital O”. In other words, to write your padding you will use code like `array[i] = 'X'` or `array[i] = '0'`.

With this scheme, there will *always* be padding to remove from the last block. This means that you need a way to know when the last block is being encrypted (so that you can add padding) and decrypted (so that you can remove padding). After the last block is decrypted, remove any trailing '0' bytes (if they exist) and then the last 'X' byte (which must always exist in the last block). (This method was suggested by Schneier and Ferguson.)

Padding Examples:

In these examples, assume a four-byte block.

- Final block: “ABCD”. There is no room for padding, so add a full padding block “X000” to the plaintext before encryption. On decryption, the second to the last block will decrypt “ABCD”. The final block will decrypt “X000” and should be dropped completely (i.e., not written to disk).
- Final block: “00”. Pad the last plaintext block before encryption so that it reads “00X0”. After decryption, “00X0” will be the last block. Remove any trailing 0s and the final X so that it reads: “00”

- Final block: “X”. Pad the last plaintext block before encryption so that it reads “XX00”. After decryption, remove the trailing 0s and the final X so that it reads “X”.

Sample Vectors

Sample vectors (i.e., transposition box sizes and input along with correctly encrypted and decrypted data) will be provided at

```
http://www.d.umn.edu/~pahp/labs/columnar-vectors.tar.gz
```

for correctness testing. Put your `columnar.c` code into the `columnar-vectors` directory and run `./test.sh`. If you would like the script to quit on the first error, use the `-q` switch. If you would like to perform “padding only” testing (see hints, below), add the `-p` switch. See the included README.txt for more information.

Grading will use both the sample vectors and new untested vectors. Therefore, you should test your code against the sample vectors and various other types of input and keys that you choose.

Submission

Submit your source code for `columnar.c` via Moodle by the due date.

Hints & Tips

Use Helper Functions!

Your code will be much more manageable and easier to debug if you create and use helper functions rather than writing one huge function with lots of repeated code. While it might be longer, it will be easier to write and understand.

For example, my solution uses the following functions:

Padding and Unpadding Blocks

```
int pad_buffer(char *buffer, unsigned int bufsize,
                      unsigned int rbuf_index)
int unpad_buffer(char *buffer, unsigned int bufsize)
```

`pad_buffer()` takes as input a `buffer` of size `bufsize` bytes where `rbuf_index` indicates the first empty byte in the buffer. It writes an 'X' into `buffer[rbuf_index]` and fills the remaining bytes with '0' characters. `unpad_buffer()` removes padding from `buffer` if it exists. Both functions modify (in place) the memory pointed to by the pointer `buffer` and return the number of padding bytes added or removed.

Transposing Blocks

```
void transpose_buffer(char *out, char *in, unsigned int dim)
```

`transpose_buffer` takes a `dim` by `dim` sized buffer `in` and performs the Columnar Transposition into the same-sized buffer `out`. It modifies `out` in place and returns nothing. Since the Columnar Transposition is symmetric, you do **not** need a special "`untranspose_buffer`" function.

Writing Blocks of Output

```
int dump_buffer(char *buffer, unsigned int bufsize,
                 unsigned int bytes, char *output_path)
```

`dump_buffer` takes in `buffer` and appends `bytes` bytes from `buffer` to the file named in `output_path`. It then erases `buffer` using `memset(buffer, 0, bufsize)`. It returns the number of bytes written.

Other Helper Functions

I also have helper functions to print out a block of data in ASCII and hexadecimal. You can use the %x format string to print in hex.

Transposition Methods: 2D Arrays or 1D Arrays + Transposition Algorithm

There are two main ways to handle transposition. Each has drawbacks.

The first method is to use two dimensional arrays. You can create two-dimensional arrays to create and manipulate your transposition box. Do this in C as follows:

```
char box[numcols][numrows];
```

Then, you can write into the grid by indexing into the 2D array as follows:

```
box[x][y] = symbol
```

You can read from the grid as follows:

```
cipher = box[0][0]
```

... which would be the "top left" character in the transposition box.

When filling the box, write rows in order, When performing the transposition, you will read the columns instead. You can "loop" your array indexes using the box dimensions and modular arithmetic.

The drawback to 2D arrays is that it is not straightforward to pass them to functions, which means that it is less easy to use helper functions in your code.

The second obvious approach is to determine the swapping pattern for the columnar cipher for a box dimension of n. Then, you can swap characters from one 1D array to new positions in another 1D array. In C, it is very easy to pass arrays by passing a pointer, so it is easy to make helper functions to make your code simpler. However, to do this you must first determine the transposition algorithm! Also, the 1D solution can be somewhat slower than the 2D one.

Finding the Length of the Input File

While you can write perfectly correct code that does not ever explicitly check the length of the file, you might find it helpful to know when you're about to reach the end of your input. C provides functions for finding the length of a file in bytes. Here's an easy way to do it. First, make sure you include the following headers at the beginning of your program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

We're going to use the stat() function, which fills a 'stat' structure with numerous pieces of information relating to the file. (See man fstat for more information.) A structure is a fixed-size data structure with named members, which can have different types (e.g., char, int, etc.). All we care about is the size in bytes, which is the 'st_size' structure member.
First, you need to declare a structure of type 'stat', named 'filestats':

```
struct stat filestats;
```

Then, when you call stat(), we will pass a pointer to filestats. The pointer is the memory address of the variable, which we can access using the '&' operator:

```
int err;
if ((err = stat(input, &filestats)) < 0) {
  printf("error statting file! Error: %d\n", err);
}
```

If the stat() call succeeds, the structure will be filled with the various pieces of information. We can access them by name, such as by printing out filestats.st_size, which we can treat as an integer of type long unsigned:

```
printf("Size of input is: %lu bytes\n", filestats.st_size);
```

This should help you determine when to add or remove block padding.

Converting String Parameters to Integers

The transposition box dimension will be the first parameter on the command line, stored in the character array argv[1]. However, this means that the first parameter will be a string “n” instead of an integer *n*. To convert the string (or ASCII) representation of the number into an integer, use the function atoi(), like so:

```
int dimension;
```

```
dimension = atoi(argv[1]);
```

Add and Test Using a “Padding Only” Mode

There are two challenges to this assignment: getting the transposition code right and getting the padding code right. Transposition cannot work properly without padding first working properly. Recognizing this, I added a “padding only” test mode to the test script, which you can invoke using the `-p` option. When using the “padding only” mode, the test script simply ensures that your code adds and removes the padding properly.

You could test in “padding only” mode when you have working padding code (but before you have transposition code). Or, you can add a fourth argument to your code that dynamically enables and disables “padding only” mode. For example, when executing `'encoder 4 /tmp/input /tmp/output 1'` or `'decoder 4 /tmp/ciphertext /tmp/plaintext 1'` (note the 1 at the end of each command) my solution performs padding but not transposition. Again, this is optional, but the trivial amount of extra code could save you hours of debugging.

Padding Hints

You must **always** add and remove padding to every input! If an input ends on a block boundary, you still need to add a full block of padding. However, **DON'T try to add padding to the original input file or a copy**. This is slow and inefficient. Instead, keep track of where you are in the file, and simply add padding to the final block of input (or after the final of input if necessary).

When decrypting, you should decrypt each block and only unpad the very last block (which **must** have padding).

You can easily keep track of this by getting the file length and keeping track of where you are in the file using a variable.

Code Running Slow?

Is your code running slowly? Large inputs, especially with a 1x1 box, can have poor performance, especially if debugging code is enabled. Consider disabling debugging code when running tests and when submitting your final code.