Elementary Encraption:
Implementing a Simple Product Cipher in C (Lab #5)

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

Overview

In this exercise, you will implement a Product Cipher (PC) by combining your Columnar Transposition Cipher (CT) and your Vigenère Cipher (VC) in multiple rounds. This will give you hands-on experience implementing product ciphers and coping with their aspects, including reversal of transformations, correctly implementing transformations, and more.

Background

A Product Cipher is a cipher that uses multiple transformations in series. For example, a VC and a TC can be combined as follows:

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

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.

Decryption can be performed using the individual decryption algorithms in reverse:

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

First, the CT is reversed, and then the VC is undone.

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:

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

we can express two rounds as:

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

we could further “collapse” this function like so:

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

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

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

we can express two rounds as:

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

… and we can further collapse this function too:

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

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). You will implement both encoding and decoding mechanisms.

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.
5. You should **NOT** use other online examples of Columnar Transposition, padding, or the Vigenère Cipher as a coding template; if you do not understand how they work, talk to your instructor or TA.
6. **The number of rounds will be read from the command line** (like dimension was for Lab 4).
7. **Keys will be read from a file** as in Lab 3.
8. **All keys will be 16 bytes in length.**
9. The dimension of the transposition box should be hard-coded to 4 (for a block size of 16 bytes).
10. Input will **not** be limited to text only (i.e., test using binary data like applications or images).
11. 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 rounds to use, e.g., 1 or 5 or 10, etc. (the number of rounds may be arbitrary)
2. A path to a key file, e.g., “keys/key_01”
3. A path to an input file, e.g., “input/file_01”
4. A path to an output file, e.g., “output/file_01_encoded”

Upon running:

./encoder 5 keys/key_01 /tmp/input /tmp/output

...the file 'output' should contain the contents of input as encoded using a 5-round VC-TC product cipher as described above. Upon running:

./decoder 5 keys/key_01 /tmp/encoded /tmp/decoded

... decoded should contain the contents of encoded as decoded using a a 5-round VC-TC product cipher as described above.

Padding and Block Ciphers

Padding is still required for this cipher, and the same scheme should be used for this lab as for the Columnar-only lab.

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, and then
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 file). 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 TC:

```
BC
DE → BDCE

FX
00 → F0X0
```

… giving us a Round 1 output of:

`BCDEFX00`

Round 2

Picking up with our Round 1 output:

`VC(11111111, BCDEFX00) → CDEFGY11`

Then, another TC:

```
CD
EF → CEDF

GY
11 → G1Y1
```

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

```
X0
00 → X000
```

… giving us a final output of: `CEDFG1Y1X000`, which is much larger than our input and, as promised, the last padding block is unciphered. This is not a worthwhile approach.

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 TC:

```
BC
DE → BDCE
```

Block 1, Round 2

`VC(1111, BDCE) → CEDF`

Then we would do a TC:

```
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:

`EX00`

Now we can do our rounds:

`VC(1111, EX00) → FY11`

Then we would do a TC:

```
FY
11 → F1Y1
```

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, F1Y1) → G2Z2`

Then we would do a TC:

```
G2
Z2 → GZ22
```

… giving us a final output of: `CEDFGZ22`. This output is much shorter and leaks much less information.

To summarize, read in 16 bytes of input. If you have a full block, perform Vigenère and Columnar in sequence for *n* rounds and write it to disk. There are two cases for when you run out of input: you might have a partial block of input or your input may fit perfectly in the last block. In the first case, fill out the remaining bytes with padding. In the second case, create a full block of padding. Then, perform Vigenère and Columar in sequence for *n* rounds and write it to disk.

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:

```
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.

```
void vigenere_buffer(char *buffer, char *keybuf, unsigned int bytes,
                   int mode)
```

`vigenere_buffer` takes a `buffer` and key (`keybuf`) of length `bytes` in addition to a `mode` parameter (to indicate encoding or decoding). If encoding, the function performs a positive shift based on the values of `keybuf`. If decoding, a negative shift. The buffer is modified in place and there is no return value.

```
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.

There's no straightforward way to pass two-dimensional arrays in C. Your choices are to pass normal arrays and construct two-dimensional arrays inside the function, or to figure out how to perform the transposition in a single array. (There's a general algorithm to do it, but it takes some thinking to figure it out.)

```
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.

Hint: In my solution, I use a while loop with *fgetc()* to read bytes from the input file, just as in the other labs. However, instead of reading bytes into a matrix or directly transposing them, I first read *blocksize* bytes into *read_buf*, a read buffer the size of the block. *read_buf* is a regular (one-dimensional) character array. I also declare and use the variables *buf_size* (the size of the buffer) and *buf_index* (the variable I use to "walk" the buffer as I write characters into it, e.g., '*read_buf[buf_index] = symbol*').

If *read_buf* isn't full inside the while loop, I don't need to do any kind of transformation; I just keep reading bytes into *read_buf*. When it is full (i.e., *buf_index == buf_size)*, I then start my multiple rounds of VC and TC, reading from *read_buf* instead of the file. I then reset *buf_index = 0* so that the buffer will be ready for the next block when the while loop starts over.

Naturally, when I hit EOF in my while loop, I'll be kicked out of the loop. However, *read_buf* and *buf_index* will contain whatever bytes that were read into *read_buf* and the position of the first empty array index before the end of file was encountered. If *buf_index* is 0, I know that the file ended on a block division and I will pad the entire *read_buf* buffer. If *0 < buf_index < buf_size*, then I know that there are *some* bytes in *read_buf* and I just need to pad to the end of the buffer. (In the case of our padding example above, only the letter 'E' would be in *read_buf* when EOF was hit, and *buf_index* would be set to 1 – the first empty space in *read_buf*.)

Once the final *read_buf* is padded, I can perform the same sequence of transformations on this last block by repeating my transformation code or (better yet), passing *read_buf* and any other necessary information to functions that perform the transformations.

Sample Vectors

Sample vectors (i.e., various round settings, keys, and input along with correctly encrypted and decrypted data) will be provided at http://www.d.umn.edu/~pahp/CS4821/labs/product-vectors.tar.gz for correctness testing. 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.

Short Answer Questions

Compression as an Entropy Estimator

While encryption attempts to reversibly encode data with a maximum of entropy, compression attempts to reversibly encode data using a minimum of entropy. In other words, compression attempts to encode data using the least number of bits possible. Therefore, you can *estimate* the entropy of a set of data by attempting to compress it with an good algorithm like ZIP or gzip. If entropy is high, the data will not compress very much (or might even expand during compression). If entropy is low, the data will compress significantly owing to the large amount of redundancy in the data.

If *F* is the size of a file when not compressed, and *C* is the size of the file when compressed, we say that the *compression ratio* (CR) achieved by the algorithm is *C/F*. For example, if the file was 4 megabytes before compression and 3 megabytes afterwards, we say that the CR is 0.75 because it is ¾ its original size. If the file expands (i.e., *C* is larger than *F*), then the CR will be greater than 1. Normally, when compressing files, we are trying to save space, so we desire the lowest possible CR. However, when compressing encrypted data as a means of estimating entropy, we would like to see a large CR, which would indicate that the ciphertext was difficult to compress.

To compress a file using gzip, execute the following command:

```
gzip -k filename
```

This will compress *filename*, creating *filename.gz*. (If you do not use the -k switch, gzip will remove the uncompressed file). You can compare the compressed and uncompressed sizes using 'ls -l'.

```
-rw-r--r-- 1 pedro pedro 1504 Feb 23 15:10 gettysburg.aes
-rw-r--r-- 1 pedro pedro 1542 Feb 23 15:10 gettysburg.aes.gz
-rw-r--r-- 1 pedro pedro 1488 Feb 23 15:06 gettysburg.enc
-rw-r--r-- 1 pedro pedro 1264 Feb 23 15:06 gettysburg.enc.gz
-rw-r--r-- 1 pedro pedro 1483 Feb 23 15:06 gettysburg.txt
-rw-r--r-- 1 pedro pedro  747 Feb 23 15:06 gettysburg.txt.gz
```

The above listing shows six files. Highlighted in green is the length, or size, in bytes of the file *gettysburg.txt*, which is 1483 bytes. The bottom line shows *gettysburg.txt.gz*, or *gettysburg.txt* gzipped. It is 747 bytes, which tells us that gzip (with default settings) achieves a CR of 747/1483, or almost exactly 0.50. The 3rd and 4th rows show *gettysburg.txt* as encrypted using a product cipher similar to the one you will develop for this exercise, and the encrypted file as compressed with gzip. The CR for these files is 1264/1488, or almost 0.85 – a significant improvement over the unencrapted version of *gettysburg*. The first two lines show the file as encrypted with AES-128 and the encrypted file compressed using gzip. Here, the CR is 1542/1504, or nearly 1.03, meaning that the file increased in size by almost 3%.

Answer the following short answer questions in a Word- or OpenOffice-compatible document or text file and include your answers with your source code (see Submission, below).

Question 1: Why are the encrypted versions of *gettysburg.txt* so much harder to compress than the plaintext?

Question 2: Why is the *gettysburg.aes.gz* **larger** than *gettysburg.enc.gz,* and even larger than *gettysburg.aes*? What does that suggest about the difference in entropy between AES's output and our product cipher's output?

Using your product cipher and the files in *product-vectors*, encrypt the inputs *12_abe.jpg* and *11_moby_dick.txt* each with the keys *01_hexadecimal*, *02_16_zeros*, *03_16_Zs*, and *04_16-random1*. Then, compress the input files and the encrypted files using *'gzip -k filename'* Compare the unencrypted, encrypted, and compressed file sizes. **Using explicit examples (including quoting file sizes)** from the files you just encrypted and/or compressed, answer the following questions:

Question 3: Does the encrypted file size depend on the key used? Why or why not?

Question 4: Does the encrypted-and-compressed file size depend on the key used? Why or why not?

Question 5: Does the encrypted-and-compressed file size depend on the input file (*abe vs. moby_dick*)? Why or why not?

Submission

Submit your product.c source and your short answer file according to the direcitons you have received from your instructor.

Extra Credit

Building a Better Product Cipher

Thankfully, there are other transpositions and substitutions besides the very weak Columnar Transposition and Vigenère Cipher. One such transposition is a *shift*.

With a *shift*, all symbols in a buffer are moved left or right by some number of positions, with the end of the buffer wrapping around to the beginning.

For example, given the buffer “AkLp”, a positive shift of 2 would result in the buffer “LpAk”. The amount to shift can be decided by various mechanisms, such as by shifting based on the numeric value of the n^{th} symbol in the key. Naturally, the shift should be performed modulo the size of the buffer.

We have already spoken in class about using *XOR* for encryption, which is a kind of a substitution transformation (symbols do not move positions, but they are changed in some reversible way). Many ciphers XOR the key (or part of the key) with intermediate data during a round.

Whether a cipher uses shifts, XORs or other transformations, they must be reversible, and their use in encoding and decoding must be carefully coordinated.

For this extra credit project, you will need to enhance your Lab 5 Product Cipher by adding **at least two additional transformations** (substitutions or transpositions). Your code **must properly be able to encrypt and decrypt the product cipher sample vectors to ensure correct operation**. Furthermore, when compressed, the output of your improved encoder must be larger than your required encoder for Lab 5. In other words, your improved encoder must increase the entropy of its output enough to be less compressible, on average, than the “standard” product cipher for this assignment. I will measure this by comparing the compressed total size of the “correct” answers against the compressed size of your output in the ENCODED directory. (You can use `./test.sh -E` to skip the correctness check and produce all encrypted output.)

Completion of this task will result in 2% of extra credit points being added to your grade, *after* all assignments, exams, etc. For example, if you end up with an average of 89% on all regular projects, completing this task will push your final average to 91%. **The deadline for this opportunity is the last day of finals week.**

Finally, as I receive submissions for this extra credit, I will measure the performance and compressibility of the submissions. The fastest encoder/decoder that improves entropy over the required code for Lab 5 will receive an additional 0.5%. The best encoder (with the highest entropy) will also receive an additional 0.5%. **Remember that it is against the rules to reuse code from existing encryption systems – you may take ideas from real ciphers, but this must be your own work.**