Assignment 7: Encrypter

CSE30: Computer Organization and Systems Fall 2021
Instructors: Bryan Chin and Keith Muller
Updated: Tuesday, November 9th, 2021 @ 5:30PM
Due: Monday, November 15th, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. **Start early**. **You MUST run the assignment on the pi cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on Edstem.

Table of Contents

Learning Goals

- Programming in ARM assembly
- Working with bit manipulation (bitwise operators, bit shifting/masking)
- Coordinating a program with both ARM assembly and C code

Assignment Overview

It has been about 6 weeks since you created the checksum program to prevent others from creating fake messages in your town. It has been a great success, you've been able to determine which signs are genuine and which signs are not. However, you've now encountered the problem of random people being able to read messages that should only be between members of the town! The location of the resource vault has been tracked, and all of your chests have been ransacked. Your town decides on the method of migrating extremely far away, leaving behind a sign with coordinates, and a checksum to prove that the message is legitimate. Obviously not fixing the problem, you arrive to your town looking like this:

I think you need to come up with a better way to share messages.

For this assignment, you will be creating a message encryption program. Using an algorithm, it will procedurally encrypt each character in a message by performing operations on the bits of the character. Additionally, this algorithm will take another character, a key, and use it to further encrypt each character.

This key will be obtained from a bookfile, which essentially is a large buffer of characters to obtain relatively random character keys from. There will be the ability to choose where to start obtaining keys from in the bookfile. Once a message is encrypted, the program will also be able to decrypt the message by performing the operations used to encrypt in reverse. This means that the offset chosen for where to start reading in the bookfile will be needed to decrypt the message as well.

Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See Edstem FAQ or watch CSE30 Development Tools Tutorial
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following GitHub repository.

1. Clone/download the GitHub repository.[1]
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/encrypter-a7-ref`.
You can directly use `encrypter-a7-ref` as a command.

Makefile: The `Makefile` provided will create a `encrypter` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead. Run `make clean` to remove files generated by `make`.

[1] If you use your own Github repo, make sure it is private. Your code should never be pushed to a public Github repository. **A public Github repo is an AI violation.**

How the Program Works

The program shall take a decrypt or encrypt flag, a bookfile flag and a file path for a bookfile, and an encryption file path. It also takes the optional argument of an offset flag and the offset.

Inputs:

- Decrypt flag OR encrypt flag
- Bookfile flag and file path of the bookfile
- Optional: Offset flag and offset
- Encryption file path

Outputs:

- If encrypting, nothing should be printed to `stdout`, only to the `encryption_file`
- If decrypting, the decrypted text should be printed to `stdout`
- If the decrypt flag and encrypt flag are both missing, print usage
- For all other misinputs, error strings are not tested, but should return `EXIT_FAILURE`

Program Arguments

Format for calling this executable with arguments (the flags cannot be grouped with each other):

```
./encrypter (-d|-e) -b <bookfile> [-o <offset>] <encryption_file>
```

Argument(s)	Description
`-d`	Sets the program to decrypt. **<u>Exactly</u> 1 of `-d` OR `-e` must be provided, but <u>not</u> both.**
`-e`	Sets the program to encrypt. **<u>Exactly</u> 1 of `-d` OR `-e` must be provided, but <u>not</u> both.**
`-b bookfile`	The path to the input bookfile (more info on this file later).
`-o offset`	`offset` is an integer specifying the number of characters to offset from the start of the bookfile to use for encryption (more info on the calculation later).
`encryption_file`	If **encrypting**, this is the path to the **output** file (overwrite whatever exists). The input text to be encrypted is passed in through `stdin`. (Highly recommend redirecting input from a file using `"<"`). If **decrypting**, this is the path to the **input** file (contains an encrypted file). The decrypted text should be printed to `stdout`. (Highly recommend redirecting output to a file using `">"`).

You **will** have to write the option parsing yourself (you may use `getopt()`). There is no loop or boilerplate code provided for option processing. Usage should only be printed if the decrypt flag or encrypt flag are missing. Otherwise, an error can be handled in any way (natural errors from function calls or error strings), but should exit the program with `EXIT_FAILURE`. Any error string returned from the reference executable is only to serve as an example for handling a misinput.

Output Examples

(These examples will use `"cat"` as the contents of files will need to be seen.)

```
$ cat original_file
The next message will contain a new-line.
$ ./encrypter -e -b book encrypted_file < original_file
< no output > // DO NOT PRINT THIS (will not be in next examples)
$ cat encrypted_file
2▒"l▒#EReV9dWŮ▒pZ▒▒,6ↄn{g▒\▒▒ℸ"▒
// There is no '\n' at the end of the line above
$ ./encrypter -d -b book encrypted_file
The next message will contain a new-line.
```

```
$ cat original_file
The offset to decrypt the next message is 12345.
But it won't contain a new-line.
$ ./encrypter -e -b book encrypted_file < original_file
$ cat encrypted_file
▒3"▒R83w5▒qf9P▒o"p+▒5i▒▒+o▒%=UoVp▒_wap,=▒▒({▒>▒▒3gQu▒▒W▒▒KX"▒?▒▒ \▒▒
// There is no '\n' at the end of the line above
$ ./encrypter -d -b book encrypted_file
The offset to decrypt the next message is 12345.
But it won't contain a new-line.
```

```
$ cat original_file
The next message will contain a new-line.
$ ./encrypter -e -b book -o 12345 encrypted_file < original_file
$ cat encrypted_file
$▒"g▒v▒!"▒>R{%vW▒pS▒▒▒mdv▒8▒趯0▒▒
// There is no '\n' at the end of the line above
$ ./encrypter -d -b book -o 12345 encrypted_file
The next message will contain a new-line.
```

In the next examples we will be using `hexdump` to see the actual bytes of the files. This will provide a better analysis of the output of the encryption, as it will show the actual values that are outputted from the encrypt function. We will also use the `"-C"` flag as it separates it into each individual byte and shows the characters (by default it only outputs hex values).

```
$ cat original_file | hexdump -C // Eight new-lines
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
$ ./encrypter -e -b book encrypted_file < original_file
$ cat encrypted_file | hexdump -C
00000000  f4 c8 c5 80 e1 c4 d6 c5
|........|
00000008
$ ./encrypter -d -b book encrypted_file > decrypted_file
$ cat decrypted_file | hexdump -C
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
```

```
$ cat original_file | hexdump -C // Eight new-lines
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
$ ./encrypter -e -b book -o 1 encrypted_file < original_file
$ cat encrypted_file | hexdump -C
00000000  c8 c5 80 e1 c4 d6 c5 ce
|........|
00000008
$ ./encrypter -d -b book -o 1 encrypted_file > decrypted_file
$ cat decrypted_file | hexdump -C
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
```

```
$ cat original_file | hexdump -C // Eight new-lines
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
$ ./encrypter -e -b book -o 12345 encrypted_file < original_file
$ cat original_file | hexdump -C
00000000  c1 d4 d4 c5 d2 80 cf c6
|........|
00000008
$ ./encrypter -d -b book -o 12345 encrypted_file > decrypted_file
$ cat decrypted_file | hexdump -C
00000000  0a 0a 0a 0a 0a 0a 0a 0a
|........|
00000008
```

As you can see in the first and second example, the encrypted file has seven of the same characters (`c8 c5 80 e1 c4 d6 c5`), offset by one byte. This will be explained later in the Obtaining the Key section and the Encryption Algorithm section.

Error Examples

```
$ ./encrypter
Usage: ./encrypter (-d|-e) -b <bookfile> [-o <offset>] <encryption_file>
```

```
$ ./encrypter -d
Unable to open book file: (null) // This string is not necessary
```

```
$ ./encrypter -e
Unable to open book file: (null) // This string is not necessary
```

```
$ ./encrypter -d -b
./encrypter: option requires an argument -- 'b' // Output from getopt()
```

```
$ ./encrypter -d -b book
Unable to open file: (null) // This string is not necessary
```

```
$ ./encrypter -d -b book -o
./encrypter: option requires an argument -- 'o' // Output from getopt()
```

```
$ ./encrypter -d -b book file_does_not_exist
Unable to open file: file_does_not_exist // This string is not necessary
```

Obtaining the Key

If you do not know what a key in cryptography is (specifically, symmetric cryptography), it is essentially similar to how a physical key and a lock works. The key is used to close the lock (or encrypt a message), and the same key is required to unlock the lock (or decrypt the encrypted message). In practice, this almost always uses the exclusive-or operation, or XOR. This is because XOR has the wonderful identity and self-inverse properties, meaning that $A \oplus 0 = A$ ($\oplus$ is the XOR symbol), and $A \oplus A = 0$. Thus, if we have the message M and we XOR it with key K, we can XOR the key again to reobtain M.

Example: $M \oplus K \oplus K = M \oplus (K \oplus K) = M \oplus (0) = M$

The bookfile is simply a file to obtain keys from. This is based on book ciphers, in which the plaintext of a book is used as a key to a cipher. This is more convenient than carrying around specific keys, as books are public and easily accessible. In the starter code, the bookfile is just a plaintext file of The Adventures of Sherlock Holmes by Arthur Conan Doyle.

How the program obtains keys from this bookfile is simply by opening it and reading characters from the file. If no offset is given, the first key should be the very first character of the file, which in the case of the starter code is `'T'` (the first line of the file being `"The Adventures of Sherlock Holmes"`). However, the next time we obtain a key, we will increment the location by one byte, meaning that the next key would be `'h'`. If an offset is given, **only the starting location** will be offset that many bytes/characters. So if the offset was 5, the first key would be `'d'` (with 0-indexing). The next key for that offset would be `'v'`, and so on and so forth.

Encryption Algorithm

There are two main steps to the encryption algorithm: swapping halves and XORing the key.
Note that the encryption algorithm is performed solely on a single character.
This is equivalent to a single byte, or 8 bits.

Step 1: Swapping Halves

Let's begin with the example of encrypting the letter `'a'`. In ASCII, `'a'` has the decimal value of 97, or hexadecimal value of 61. This means in binary it is the following (6 = 0110, 1 = 0001):

0	1	1	0	0	0	0	1

The first step of the algorithm is to switch the first four bits with the last four bits, keeping the order of these bits. After this procedure, the result will be the following:

0	0	0	1	0	1	1	0

This process can be done any number of ways. More notes later in the implementation section. However, one thing to note is that at this point in the process, the encryption would result in the ASCII character of value 22, which is not a printable character.

Step 2: XORing the Key

Once we have performed the first step, a single bitwise operation needs to be done. Using a key (how to obtain the key is described above), the next step of the algorithm is to exclusive-or (XOR) it with the result from the first step.

For example, let's use the letter `'T'` as the key. In ASCII, `'T'` has the decimal value of 84, or hexadecimal value of 54. This means in binary it is the following (5 = 0101, 4 = 0100):

0	1	0	1	0	1	0	0

XOR 0101 0100 with 0001 0110, and obtain the final result:

0	1	0	0	0	0	1	0

This gives hexadecimal 42, or decimal value 66. The corresponding character in ASCII is `'B'`. Thus the outputted encrypted character would be `'B'`.

As written in the Obtaining the Key section, the actual last step of the encryption algorithm is to increment the position that you are obtaining the key from to get another key, but this should be handled outside of the encryption functions.

Program Implementation

This assignment does not have many functions to implement! There is only the main function and the `encrypt()` and `decrypt()` functions. However, these last two functions must be implemented in ARM assembly.

Allowed Instructions

You are only allowed to use the instructions provided in the ARM ISA Green Card. Failure to comply will result in a score of 0 on the assignment.

Some examples of types of instruction **NOT ALLOWED in any CSE30 assignment**: This is not an exhaustive list. For this assignment, you should not need to do any data transfer instructions.

Example	Description	Comment
`ldrb r0, [r4], #1`	post indexed addressing	use only [register, #offset] or [register] addressing
`ldr r0, [r4, r2]`	double register indirect	use only [register, #offset] or [register] addressing
`ldr r0, [r4, r2, lsl #2]`	addressing with scaling	use only [register, #offset] or [register] addressing
`mov r6, r2, lsl #2`	immediate with shift	use only simple immediates (no shift)
`movgt`	conditional move	do not use conditional instructions (only branches may be conditional)
`bleq`	conditional branch and link	use only bl

Functions and Behaviors to Implement

`char encrypt(char inchar, char key)`

Arguments:

`char inchar`	→	The char that is being encrypted
`char key`	→	The character key that is XOR'd

Operation:

- Uses bit manipulation to perform the encryption algorithm.
  - The encryption should not take more than 15 assembly instructions, if you have more, you are probably overcomplicating the algorithm.
- Make sure to not edit any other lines except the ones in between the limits
- Returns the char that results from the encryption.

You may only use up to 4 registers (R0, R1, R2, and R3). The compiler will place the value of inchar in R0 and the value of key in R1 when your function is called. The return value should be on R0 when your function completes.

`char decrypt(char inchar, char key)`

Arguments:

`char inchar`	→	The char that is being decrypted
`char key`	→	The character key that is XOR'd

Operation:

- Uses bit manipulation to perform the decryption.
  - The decryption algorithm is not explicitly described in this writeup; however, consider how one might reverse the encryption algorithm.The assembly instructions for this function are not strictly the encryption assembly instructions simply listed backwards. However, you should have near, if not the same, number of instructions in this function as encryption.
- Make sure to not edit any other lines except the ones in between the limits
- Returns the char that results from the decryption.

You may only use up to 4 registers (R0, R1, R2, and R3). The compiler will place the value of inchar in R0 and the value of key in R1 when your function is called. The return value should be on R0 when your function completes.

`int main (int argc, char **argv)`

Arguments:

< not detailed as they are the typical `main()` arguments >

Operation:

1. Process option flags (using `getopt()`).
    a. This should include handling if the arguments passed in are correct, and exiting with error/failure otherwise.
2. Open the bookfile for reading and obtain its size.
    a. Depending on implementation you may want to use any of the following: `open()`, `fopen()`, `fdopen()`, `stat()`, `fstat()`. If you don't know what these functions do or how to use these functions, please check their man pages.
3. Create a dynamically allocated array of chars with the size of the bookfile
    a. How do you determine the size of a file?
        i. Use either `fstat()` or `stat()` to fill in a `struct stat` and obtain `st_size`.
        ii. The man page has examples, so **please** check it before asking for help.
4. Read the bookfile data into the array
    a. This should be done using `fread()`. More information on this function in the Notes About `fread()` section below.
5. Open the `encryption_file` to write/read
    a. If encrypting, open the `encryption_file` for writing
    b. If decrypting, open the `encryption_file` for reading
6. Begin processing lines to either encrypt or decrypt
    a. If encrypting, use `fgets()` on stdin to obtain the lines to encrypt. Then, encrypt each character in the line, writing to the `encryption_file` after every call. This should be done using `fwrite()`.
        i. Read the man page for how to use `fwrite()`, you will only be writing a single character to the file per call.
    b. If decrypting, use `fread()` on `encryption_file` to obtain the lines to decrypt. Then decrypt each character, printing to the `stdout` after every call.
    c. When using characters from the bookfile as the key, **do not forget** to access the bookfile using the offset. This offset should be incremented by 1 after every call to encrypt or decrypt.
        i. NOTE: The passed in offset does NOT have a limitation on its size. It can be bigger than the bookfile. You will need to modulo it with the file size.
7. Close all of your files and free any dynamically allocated memory

Run Valgrind to check for memory leaks and memory errors. Valgrind is a suite of tools that do memory error analysis, among other things. Use `man valgrind` for more information. When you run `valgrind ./encrypter <args>`, it should report "All heap blocks were freed -- no leaks are possible" and "ERROR SUMMARY: 0 errors from 0 contexts".

Suggested Debugging/Development Tips

This program is not bigger than prior ones. However, we still would recommend these steps.

1. START EARLY!
2. Read the instructions carefully.
3. Check the FAQ often and search Edstem for previous questions, if possible.
4. Outline the algorithm in pseudocode. Tutors may ask to see this!
5. Be sure to check whether or not you are properly opening and reading files. This can be done by printing strings.
6. Use the example from the encryption algorithm to test your `encrypt` function. You can always test `decrypt()` by using the character obtained from the encrypt function and seeing if you get the original character back.
7. If you are working on a bug, use GDB. You can set breakpoints at interesting points (e.g. `break encrypt()` and step through your code to watch it encrypt a character). You can make better use of tutor lab hours if you have a debug session to show them. If you have a segmentation fault, run GDB before going to a tutor. Otherwise, they may not have time to help you, especially when the queue gets busy.
8. TESTING: **List** all the test cases you can think of (re-read this document). There are not as many unique test cases for this assignment, but you should still always think through test cases for every assignment you do.

   The public autograder will only test some features. DO NOT RELY ON THE AUTOGRADER (many CSE30 students have been burned by this). Test your code using your own test cases!!!

Notes About `fread()`

`fread(void *ptr, size_t size, size_t nmemb, FILE *stream).fread()` is an IO function that reads from a file descriptor into a memory buffer (`man 3 fread()`). It will read at most `size * nmemb` bytes into the memory area pointed to by `ptr`. It returns the number of bytes actually read (either `size * nmemb` or a smaller number).

In a real program, the book file is likely to be too large to read into memory all at once, but for this assignment we want you to get more pointer practice, so we are going to read the book into main memory. Again, in a real program, the book file may be too large to fit into main memory efficiently and take a long time to read the file. There also is the possibility of the file size changing, and thus you may not read in all the data if you use the size of the file before the read. Instead, it's good practice to read the file in manageable chunks.

For example, one could read the book file in chunks of 1024 bytes. [2]
Use something like:

```
// CHUNKSIZE is a #define (in this example, it is 1024)
while ((bytes = fread(file_ptr, 1, CHUNKSIZE, stream) != 0){
   file_ptr += bytes;
}
// here, one would normally test for errors using feof(), ferror()
// and clearerr()
```

In a real program, you may only need part of the book file at a time. For example, if the book file were 8 GB, you might only need to access say 64KB of the file at a time. In this case a program could allocate a memory buffer of 64KB and process 64KB of input data at a time.

In our HW assignment, you may opt to read the entire file in any chunksize (stride) you wish.

[2] In practice choosing a multiple of the device block size or machine page size maybe more appropriate:

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled "A7: Encrypter".
   You will submit the following files:

   ```
   main.c
   cipher.s
   README.md
   ```

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
   a. Checks that all required files were submitted.
   b. Checks that `main.c` compiles without warnings.
   c. Runs some tests on the resulting `encrypter` executable.

Style Requirements

Points WILL be given for style, and teaching staff won't be able to provide assistance or regrades unless code is readable. Please follow these Style Guidelines for ARM assembly.

Grading Breakdown [50 pts]

<u>Make sure to check the autograder output after submitting!</u> We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

To encourage you to write your own tests, we are not providing any public tests that have not already been detailed in this writeup.

The assignment will be graded out of 50 points and will be allocated as follows:

- **No warnings when compiling**.
- **No errors from Valgrind at any point**.
- **The public tests.**
- A variety of hidden tests that will test many different cases.
- In progress… Check back later!

This assignment will also have 5 points dedicated to style (see above ARM assembly style requirements).

NOTE: The end to end tests expect an EXACT output match with the reference binary. There will be NO partial credit for any differences in the output. Test your code - DO NOT RELY ON THE AUTOGRADER FOR PROGRAM VALIDATION (READ THIS SENTENCE AGAIN).

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster without warnings. **Any assignment that does not compile will receive 0 credit.**

Checking For Exact Output Match

A common technique is to redirect the outputs to files and compare against the reference solution[3]:

```
./your-program args > output; our-reference args > ref

diff output ref
```

If the second command outputs nothing, there is no difference. Otherwise, it will output lines that differ with a < or > in front to tell which file the line came from. Note that in this program there is FILE CREATION. Meaning, that these files should be diff'd instead of `stdout`.

END OF INSTRUCTIONS, PLEASE RE-READ!

[3] You might want to check out `vimdiff` on the pi-cluster (https://www.tutorialspoint.com/vim/vim_diff.htm).

Writeup Version

1.0	Initial release.
1.0.1	Fixed code for fread( ) example. Should be …, 1, CHUNKSIZE, … to return number of bytes read. Additionally add some clarification on parameter passing and registers in function implementation.
1.0.2	Updated output examples for the new book file.