Assignment 9: Encrypter Revisited

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Midpoint Deadline: Monday, November 29th, 2021 @ 11:59PM

Official Deadline: Thursday, December 2nd, 2021 @ 5:00PM

Submission Deadline: Friday, December 3rd, 2021 @ 11:59PM

[No support will be available on Friday, Dec 3rd]

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.

Final Survey (2 pts)

Please fill out this survey: https://forms.gle/VUWNQyeRJnjSpY887

Table of Contents

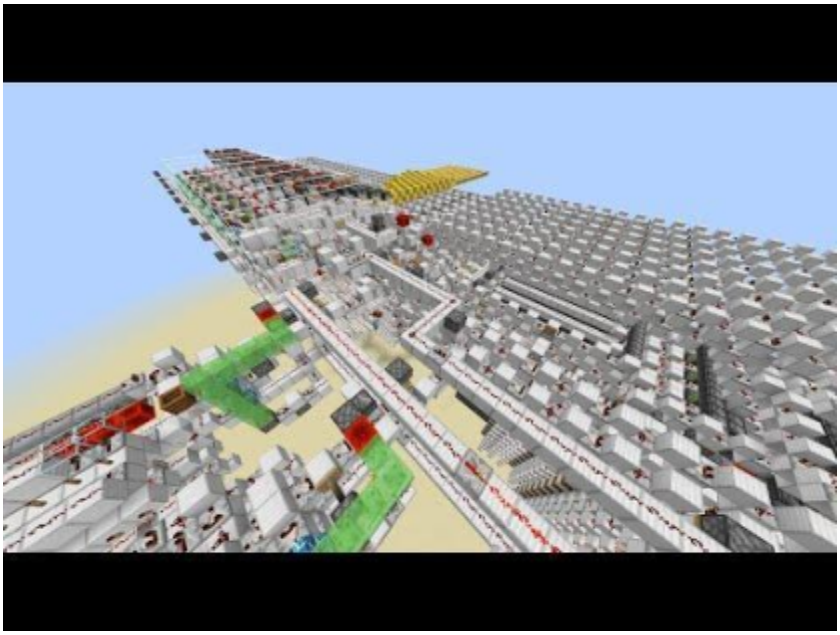
Learning Goals

- Programming in ARM assembly
    - Working with local stack variables
    - Passing pointer to functions
    - Calling functions with 6 arguments (4 in registers and two on the stack)
    - Calling C routines (stdio library functions i.e. `fread()`, `fwrite()` and `fprintf(stderr, …)` from ARM assembly functions
- Typical file sequential file processing
    - Reading, operating on the data (encrypt/decrypt) and writing
- Writing a program with an ARM `main()`

Assignment Overview

Your encrypter from assignment 7 was a smashing success! Your town no longer has any problems with communication as you have been able to send encrypted messages through files (links shared through signs). However, this sharing of files has begun to be a little tedious, and so your mayor is now requesting you to build the program in Minecraft. Is that even possible?

Fortunately, you have experience with redstone computers from assignment 3! And, with your greater knowledge of how computers run code using assembly, you better understand how to create a computer that can execute instructions in Minecraft! Or you can just copy someone else's computer:



However, to be able to run on the Minecraft computer, you will need to convert your program to something lower-level. In this assignment, you will be writing a new version of the encryption program from HW7, but this time, completely in ARM assembly. You will reuse your two encryption functions, `decrypt()` and `encrypt()` (keep `cipher.s` unchanged). However, you will be writing the `main()` function in ARM.

You will **not** have to implement the exact `main()` from HW7. One change is that you will be calling a supplied `setup()` function (written in C) that parses options and opens all the files for you. Another change is that there will be much more detailed error messages. The last main change is that the program will operate in a simpler manner than in HW7, as in this assignment you **should not do** any of the following:

1. Implement starting offsets into the book file
2. Handle wrap-around of the book file
3. Use stat() or fstat() to determine file size
4. Dynamically allocate memory with malloc() or calloc()
5. Read the book file with a single read

Besides all those changes, your main program will either encrypt or decrypt the files as before, calling your previously written `decrypt()` and `encrypt()` functions. (If you did not do HW7 or had problems with your functions, you will be responsible for their correctness in this PA).

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-a9-ref`.
You can directly use `encrypter-a9-ref` as a command.

Makefile: The `Makefile` provided will create an `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. You must make sure your code compiles/assembles without warning messages prior to turning it in. Run `make clean` to remove files generated by `make`. **There is a new target in the makefile: `test`. By running `make test` you can test your program with your own input messages and book files. The names of the files used with this target can be found in the Makefile.**

[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.

Inputs:

- Decrypt flag OR encrypt flag
- Bookfile flag and file path of the bookfile
- Encryption file path

Outputs:

- If encrypting, nothing will 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> <encryption_file>`

Argument(s)	Description
`-d`	Sets the program to decrypt. **Exactly 1 of `-d` OR `-e` must be provided, but not both.**
`-e`	Sets the program to encrypt. **Exactly 1 of `-d` OR `-e` must be provided, but not both.**
`-b bookfile`	The path to the input bookfile (more info on this file later).
`encryption_file`	When **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 `"<"`). When **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 use the provided C language function `setup()` (in `setup.c`) to handle all options processing and the opening of files. If setup returns a non-zero you should exit the program with `EXIT_FAIL` (defined in `encrypter.h`) from `main()`. If your program completes without errors, then you should return an `EXIT_OK` (defined in `encrypter.h`) from `main()`.

Output Examples

These are examples taken from A7, omitting the offset examples. Again, these examples will use "`cat`" as the contents of files will need to be seen. We've included the echo command to create the first file. (The encryption has been updated to be from bash rather than MobaXterm).

```
$ echo "The next message will contain a new-line." > original_file
$ 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 the next examples)
$ cat encrypted_file
◆3"◆2◆"l◆#EReV9dWŮ◆pZ◆◆,6↗n{g◆\◆◆⊓"◆◆
// There is no '\n' at the end and some characters are not printable
$ ./encrypter -d -b book encrypted_file
The next message will contain a new-line.
```

```
$ echo -e "The next message will have:\nOnly new-lines" > original_file
$ cat original_file
The next message will have:
Only new-lines
$ ./encrypter -e -b BOOK encrypted_file < original_file
$ cat encrypted_file
◆3"◆2◆"l◆#EReV9dWŮ◆p◆y=◆蛊◆◆q◆4◆◆>B◆
// There is no '\n' at the end and some characters are not printable
$ ./encrypter -d -b BOOK encrypted_file
The next message will have:
Only new-lines
```

In the next example, 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).

```
$ echo -ne "\n\n\n\n\n\n\n\n" > original_file
$ 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
```

Error Examples

`setup.c` will handle the error conditions from parsing the options to opening the files. However, during the processing of the data in your `main()` function (written in assembly) you should print out the following messages. We will not give you specific input files for this so that you may create the test cases yourself. Note: `EXIT_FAIL` is defined in `encrypter.h`

When the Bookfile is shorter than the message you are processing
(Print this string to stderr and exit the program returning an EXIT_FAIL)

```
.string "Bookfile is too short for message\n"
```

When the write to the output fails (or is short)
(Print this string to stderr and exit the program returning an EXIT_FAIL)

```
.string "Write failed on output\n"
```

Optional error messages (not checked for assignment)
(Print this string to stderr and exit the program returning an EXIT_FAIL)

```
.string “Read failed on input\n”
.string "Read failed on bookfile\n"
```

Operation of `main()`

`main()` works by processing the input file in a sequential manner. It reads a block of bytes into an input/output buffer. Next it reads up the **same number of bytes** into the book buffer. If there are not enough bytes in the book file to match the number of bytes you read from the input file, you will then print an error message `"Bookfile is too short for message\n"` to stderr and exit the program, returning an `EXIT_FAIL`.

When doing I/O you should never expect to always get all the bytes you requested in an `fread()` call[2]. In this program you have to encrypt/decrypt each byte read with a byte read from the bookfile. So if you read `readcnt` bytes from the input file, you will need to read `readcnt` bytes from the bookfile to process the bytes read from the input file.

The typical approach to processing data in this manner (what you will need to code) uses two buffers, one for the input to be processed and one for the bookfile that contains the encryption keys. Each byte in the input buffer is matched by a corresponding byte in the bookfile buffer. The input file and book file are read and then processed block at a time, reading in sequential order from the first byte until EOF is reached on the input file. So both buffers contain the same number of bytes read from their corresponding files. If the bookfile is too short (you reach EOF on the bookfile before the input file), there will not be enough encryption keys to process the input message, so you will write an error message to `stderr`, remove any output (see `encryptdelete()` below), and exit the program returning an `EXIT_FAIL`.

In order to keep the two buffers synchronized (so they contain the same number of bytes) you might write a loop similar in functionality to the one shown below. The example is shown in C, you will write your version in assembly.

[2] For example, suppose we get 4096 bytes from reading `FPinout`. Then we call `fread()` on `FPbook` and ask for 4096 bytes but suppose we only get 2048. So we'd set `toread` (see C code below) to 2048 and try again. We keep doing this until we get all 4096 bytes from `FPbook` or we encounter EOF.

In this example: `FPin` is the input `FILE *`, `FPbook` is the bookfile `FILE *` and `FPout` is the output `FILE *`. You can depend on `fread()` to return a value that is never larger than the `toread` or `BFSZ` parameters (0 return is either EOF or error; we will ignore error conditions here).

```
  while ((readcnt = fread(iobuf, 1, BFSIZ, FPinout)) > 0) {
    pos = 0;
    toread = readcnt;
    // read book until we have read readcnt bytes or encountered EOF
    while ((bytes = fread(&bookbuf[pos], 1, toread, FPbook)) > 0) {
        if ((pos = pos + bytes) == readcnt)
                break; // we got all the bytes we needed
        toread = toread - bytes;  // short read, read what we need
    }
    if (bytes == 0){
        // either reached EOF or an error; we assume it is EOF
        // print message, break out of the loop, if encrypting
        // delete any output and exit with EXIT_FAIL
        fprintf(stderr, "Bookfile is too short for message\n");
        break;
    }

    // en/decrypt each char in iobuf[j] with bookbuf[j]
    // write out encrypted buffer
    if (fwrite(iobuf, 1, readcnt, FPout) != readcnt) {
            fprintf(stderr, "Write failed on output\n");
            break;
           // write failed, if encrypting clean up any output,
          // print message
          // break out of the
          // loop and exit with EXIT_FAIL
    }
    // go back and process another block of data
}
```

Notice how the starting address of `bookbuf` is adjusted forward by the number of bytes read so far. In the example above, we are assuming the calls to `fread()` are functioning properly and a return of 0 simply indicates an `EOF` situation. In a real situation you would use additional code to differentiate between an `EOF` from an error.

Be aware that when writing `main()` that even when using all the preserved registers you will need to be careful in how you use them since there are a lot of function calls in `main()` and you will quickly run out of preserved registers. You can allocate additional variables on the stack, but accessing them will be slower than registers and each access requires around two instructions. In the loops above notice the use of break to get out of loops. This was written this way in an attempt to be frugal in the use of flags and other loop control variables. Using flags and other constructs for loop control will consume registers and may force you into having to write a lot of code to move variables between stack memory and registers (as an aside this is called register spill and register fill).

Say we have two 1024 byte buffers: `iobuf[1024]` and `bookbuf[1024]`. When we call `fread()` to fill the entire iobuffer buffer (all 1024 bytes) and `fread()` returns the six (6) characters SAMPLE (the return value from `fread()` is 6). We would then call `fread()` for the book buffer and fill it with just six (6) characters `THEADV`.

Input/output buffer

S	**A**	**M**	**P**	**L**	**E**		

Book buffer

T	**H**	**E**	**A**	**D**	**V**		

Now we will encrypt in place. We replace each char in the `iobuf` with its encrypted (or decrypted) version. Let the variable `readcnt` be the number of characters read from the input file = 6;
Next, you can process the `iobuf` with an inner loop like (encryption shown for example):

```
for (int j = 0; j < readcnt; j++) {
      iobuf[j] = encrypt(iobuf[j], bookbuf[j]);
 }
      /* write out iobuf as it has now been encrypted */
```

Program Implementation

In this assignment you only have to write the `main(int argc, char **argv)` and `encrypt()/decrypt()` functions in assembly. You may reuse your `cipher.s` file from HW7 unchanged. You will use the provided `setup.c` file to handle options and file processing.

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, 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 Provided

`int setup(int argc, char *argv[], int *mode, FILE **BOOK, FILE **INPUT, FILE **OUTPUT)`

Arguments:

`argc`	→	argc copy from main (passed in r0)
`*argv[]`	→	argv copy from main (passed in r1)
`*mode`	→	pointer to integer stack variable (1 = encrypt, 0 = decrypt) (passed in r2)
`**BOOK`	→	pointer to FILE * pointer stack variable (passed in r3) bookfile
`**INPUT`	→	pointer to FILE * pointer stack variable (arg5 on stack) input file
`**OUTPUT`	→	pointer to FILE * pointer stack variable (arg6 on stack) output file

Operation:

- Setup uses `getopt()` to parse the option flags for you. It will then open and return to you the three file pointers you will use in main.
- `mode` contains a 1 when there is a `-e` (encrypt) and a 0 when there is a `-d` (decrypt)
- `BOOK` contains an open for read `FILE *` pointer to the bookfile
- `INPUT` contains an open for read `FILE *` pointer to the input for the command (`stdin` for encrypt or the encryption file for decrypt)
- `OUTPUT` contains an open for write `FILE *` to the output for the command (encryption file of encrypt or `stdout` for decryption).
- Setup returns an `EXIT_OK` on success or an `EXIT_FAIL` if it fails. If `setup()` fails you exit the program, returning `EXIT_FAIL`. On a successful return from `setup()` you process the files.
- Please take notice that `setup()` has six (6) arguments, so four (4) will be passed via registers and the remaining two (2) will be passed on the stack.
- Four (4) of the arguments are output parameters that point to variables allocated on the stack.

`void encryptdelete(void)`

Arguments:

None

Operation:

- `encryptdelete()` removes the `encrypted_file` being written that is not complete **only when operating in `ENCRYPT_MODE`, (it should do nothing in `DECRYPT_MODE`).**

Usage:

- You will call `encryptdelete()` at the end of `main()`, after `fclose(INPUT)`, `fclose(BOOK)`, `and fclose(OUTPUT)`. See the conditions for when to call `encryptdelete()` in the section below titled: **Exiting main()**

Function to Write

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

Arguments:

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

Include file: `.include "encrypter.h"`

`encrypter.h` contains the definitions for the `mode` values and the return value for `setup()` that have to be the same for `main()` and `setup()` (both include the same `encrypter.h` file).

Operation:

1. Allocate local variable space on the stack for `mode`, `BOOK`, `INPUT`, `OUTPUT`. This is necessary since these are output parameters in the call to `setup()`; they must reside in memory for `setup()` to be able to change their contents. All four are pointers so they need 4 bytes each (16 bytes total).
2. Allocate space on the stack for two buffers (each 1024 in size). One is for `iobuf` where the input will be read into and the other is for `bookbuffer` for the corresponding keys.
3. Allocate space on the stack for any additional local variables you need when you run out of registers.
4. Map out how you are going to use the preserved registers assigning them as local variables. You may want to keep frequently referenced local variables in preserved registers. Any variables that you need to use across function calls (so they cannot be in temporary registers) will have to be either in preserved registers or stored on the stack to protect their contents.
5. Call `setup()` and it will open the files and place copies of the open `FILE *` pointers in each of the local stack variables: `BOOK`, `INPUT`, and `OUTPUT`. It will place `ENCRYPT_MODE` in the stack variable mode if you are encrypting (`-e` flag) or `DECRYPT_MODE` if you are decrypting (`-d` flag).
    a. You will either continue processing or exit if `setup()` fails. If `setup()` fails, `setup()` will print all the necessary error messages for you, then follow the instructions below on exiting `main()`.
6. Read a block of characters from the input file remembering how many bytes you read. If `fread()` returns a 0 you are done, following the instructions below for exiting `main()`. A 0 return from `fread()` may also indicate an error while performing the read; however, you do not have to handle this situation in your code.
7. Read the same number of bytes into `bookbuffer` (look at the sample on how to process files sequentially in sync). If the bookfile has less characters than the input file, you print the line to `stderr` (use `fprintf()`) `"Bookfile is too short for message\n"`, then follow the instructions below for exiting main()
8. Encrypt the bytes in the input buffer one character at a time, replacing each entry in the iobuf with its encrypted (or decrypted) value you get from calling your decrypt() or encrypt functions.
9. Write the processed iobuf to the `OUTPUT` File (`fwrite()`) and go back and read another buffer (step 6 above).

Exiting main()

Close the three (3) IO streams by calling `fclose()` for each of `INPUT`, `OUTPUT` and `BOOK`.

When you are exiting your program because of an error, you will return `EXIT_FAIL`.

Otherwise you will return an `EXIT_OK` (both are defined for you in `encrypter.h`).

In addition, when exiting from an error and the program is running in ONLY ENCRYPT_MODE, you need to call `encryptdelete()` after closing the three (3) IO streams when:

1. **A short read on the bookfile**
2. **A failure to write the output**
3. **You had an `EXIT_FAIL` return from `setup()`**

Functions Reused from HW7 (Need to Write if You Skipped HW7)

`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 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

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”.

The starter code for main.s contains a suggested minimum stack frame that includes variable space and argument space.

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. Make sure you understand how `setup()` works. You can write a simple `main()` in C and call it to see what it does.
7. If you are working on a bug, use GDB. You can set breakpoints at interesting points (e.g. right after the return from `setup()` to check that you know how to use the output variables and you passed them properly).You can make better use of tutor lab hours if you have a debug session to show them. If you have a segmentation fault, debug with 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!!!**

Midpoint Checkpoint

This part of the assignment is due earlier than the full assignment, on Friday 11/29 at 11:59pm. There are no late submissions.

Complete the Gradescope assignment “A9: Midpoint”, an Online Assignment that is done entirely through Gradescope. This assignment consists of 3 short questions about this writeup, and a free-response question where you will document your `main()` algorithm and stack frame. This should be either a pseudocode or a C version of `main()`. This is a planning document and does not need to reflect your final implementation, although you are encouraged to keep it up to date.

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled "A9: Encrypter Revisited".
   Please submit only the files mentioned below and ensure to avoid submitting the compiled objects and executables.
   You will submit the following files:

   ```
   main.s
   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.s` assembles without warnings.
   c. Runs some tests on the resulting `encrypter` executable.

Style Requirements

Points WILL NOT be given for style, but 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 [48 + 2 pts]

Make sure to check the autograder output after submitting! 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:

- **Final survey:** 2 points. Submit here: https://forms.gle/VUWNQyeRJnjSpY887
- **Midpoint writeup:** 5 points.
  This part of the assignment is due earlier than the full assignment, on Monday 11/29 at 11:59pm. Complete the Gradescope assignment "A9: Midpoint".
- **No warnings when compiling.**
- **The public tests.**
- A variety of hidden tests that will test many different cases.

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`.

You can also encrypt a file with your program and decrypt it with the reference program to see if you get the original message back (and the reverse test as well: encrypt with the reference program and decrypt with your program.

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.