Bryan Lee ID: 922649673
Github: BryanL43 CSC415 Operating Systems

Assignment 6 – Device Driver

Description:
This program is a simple device driver that encrypts or decrypts a message using Vigenere's cipher, which runs on Linux. The program also accepts a custom key to encrypt/decrypt the message, offering more variety to the output.

How to load the device driver:

1. Navigate to the 'Module' directory.
2. Run 'make' in the Linux terminal.
3. If compiling the driver for the first time, the "install.sh" file needs its permissions changed using "chmod +x ./install.sh" without quotation marks.
4. Call the shell-install script by inputting "./install.sh" without quotation.
    a. Input password if required.
5. Navigate to the 'Test' directory.
6. Two options to <u>interact</u> with the device driver:
    a. make run
        i. Encrypts a predefined test message
    b. make
        i. A general compilation that will accept the following inputs in the Linux terminal:
            1. ./lee_bryan_HW6_main e "Message to encrypt" "key"
            2. ./lee_bryan_HW6_main d "Message to decrypt" "key"
        ii. The parameter follows this format sequentially:
            1. The file name
            2. Cipher mode: 'e' represents encrypt, and 'd' represents decrypt
            3. "Message to encrypt/decrypt"
            4. "key" that is used to encrypt/decrypt the associated message

How to unload the device driver:

1. Navigate to the 'Module' directory.
2. Run 'make clean' in the Linux terminal.
3. If compiling the driver for the first time, the "uninstall.sh" file needs its permissions changed using "chmod +x ./uninstall.sh" without quotation marks.
4. Call the shell-uninstall script by inputting "./uninstall.sh" without quotation.
5. Navigate to the 'Test' directory.
6. Run 'make clean' in the Linux terminal.

Approach:
My first step in creating the Vigenere cipher device driver is to follow along with the lecture recording to acquire the device driver's base code. The base code is necessary to establish the module script's foundation and create the open, close, 'ioctl,' 'init_module,' and

'cleanup_module' functions. The read, write, and 'ioctl' functions will differ due to the need to read/write into a buffer for encrypting or decrypting.

The next step is to edit the device driver's data structure to hold the message, key, and cipher mode. The most critical part of my design is to pass the key string into my 'ioctl' function and copy it into the data structure. Doing so allows me to separate writing the message to encrypt/decrypt and the key to avoid overwriting issues. I will then use a switch statement to determine whether the command parameter specifies encryption or decryption for the message and execute its appropriate operation.

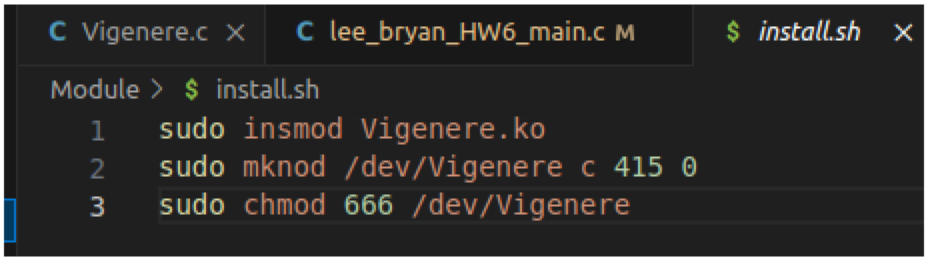
The other important step is implementing the read-and-write functions before my encryption/decryption operations. They are necessary to accept input and output data to the user's test application. Both functions will check whether the requested byte size to read/write will be greater than zero to ensure valid operations. The two functions will also need to ensure that the requested byte size does not exceed the buffer size or data size. Now, of course, per the name of the function, the write function will copy the user's message to the device driver's data structure, and the read function will do the opposite.

After creating the functions needed for the user to interact with the device driver, encryption and decryption can then be implemented. Since I chose to implement Vigenere's cipher, my design structure will follow as such for both encrypting/decrypting:

1. Resolve the key. What does this mean? If the key is less than the length of the message, then I will repeat the key to match the size of the message, which is necessary for Vigenere's cipher.
    a. To clarify: Given "This is a test message" with "key"
        i. "key" will be resolved to "keykeykeykeykeykeykeyk"
2. Encrypt/decrypt the message. I will iterate through the message and encrypt/decrypt one letter at a time. The non-alphabetical characters will not undergo any changes.
3. Encrypting/decrypting one character at a time. This is a separate operation as it is a little complex. Focusing just on encrypting, as decrypting will just be the reverse:
    a. If the character is a lowercase letter:
        i. Ensure that the letter of the associated key is lowercase.
        ii. Shift the character by the position of the key's character forward. For example: 't' shifted by 'c' will yield 'v.'
            1. Note: 'a' will be in the 0th position. So 'c' will be in the 2nd position. Thus, 't' shifted two positions forward yields 'v.'
    b. If the character is an uppercase letter:
        i. Ensure that the letter of the associated key is uppercase.
        ii. Shift the character, similar to step a.ii, but for uppercase letters.

The last step is to write the main function that will test the device driver in the "Test" directory. This will be reasonably straightforward, as I will first parse the arguments and ensure a valid cipher mode represented as either 'e' or 'd.' Next, I will need to open the device driver, write the message to it, execute the device driver with 'ioctl,' and read the response. Most of this is just simply calling the device driver's function, i.e., write, read, open.

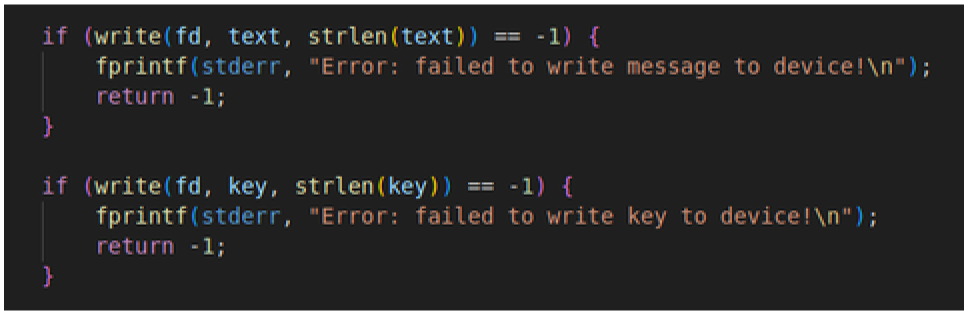
Issues and Resolutions:
The first issue I encountered was attempting to compile the device driver. I initially thought that I could simply compile it with "make run" and change the driver's permissions. However, that did not work, as I completely forgot to move and configure the necessary device driver objects into the dev folder. This was an easy fix by simply creating a shell script as follows:



```
sudo insmod Vigenere.ko
sudo mknod /dev/Vigenere c 415 0
sudo chmod 666 /dev/Vigenere
```

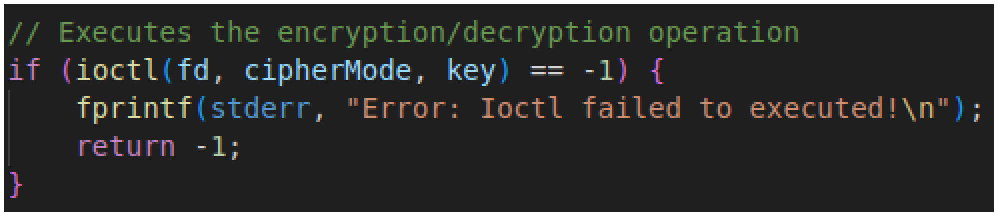
However, creating the install shell script didn't exactly resolve the compilation issue. A less notable issue was that I simply forgot to change the shell script's permissions to allow for execution. This was resolved by simply using 'chmod' as specified in the approach.

Another issue I encountered was passing a key into the device driver. I initially used the write function to write both the text and key, but that created a lot of overwriting and conflicts.
An image of the initial approach:



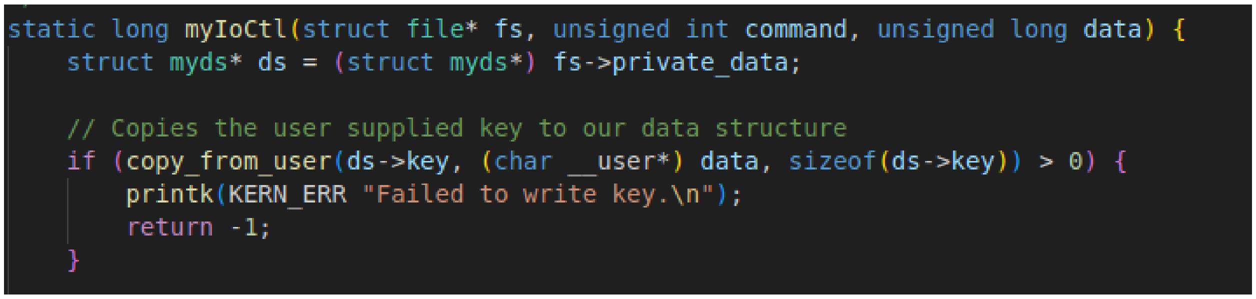
```
if (write(fd, text, strlen(text)) == -1) {
    fprintf(stderr, "Error: failed to write message to device!\n");
    return -1;
}

if (write(fd, key, strlen(key)) == -1) {
    fprintf(stderr, "Error: failed to write key to device!\n");
    return -1;
}
```

To resolve this issue, I had to pass the key through the "ioctl" function rather than attempting to write it:



```
// Executes the encryption/decryption operation
if (ioctl(fd, cipherMode, key) == -1) {
    fprintf(stderr, "Error: Ioctl failed to executed!\n");
    return -1;
}
```

However, doing so also required some changes to how I handled the function of the device driver. All I needed to do was copy the data from the user to the key buffer in the data structure, as shown in the image below:



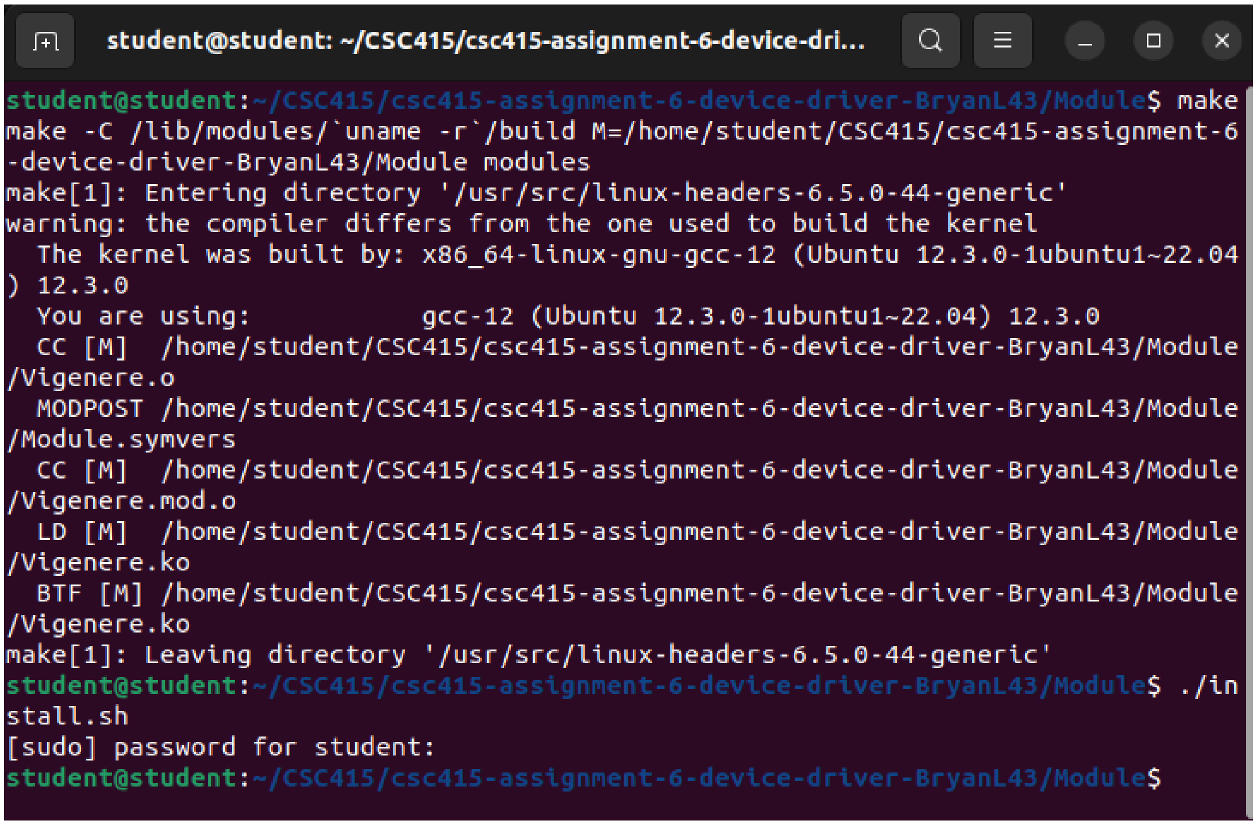
```
static long myIoCtl(struct file* fs, unsigned int command, unsigned long data) {
    struct myds* ds = (struct myds*) fs->private_data;

    // Copies the user supplied key to our data structure
    if (copy_from_user(ds->key, (char __user*) data, sizeof(ds->key)) > 0) {
        printk(KERN_ERR "Failed to write key.\n");
        return -1;
    }
```

This offered a cleaner approach and eliminated the errors I was facing.

Analysis: N/A

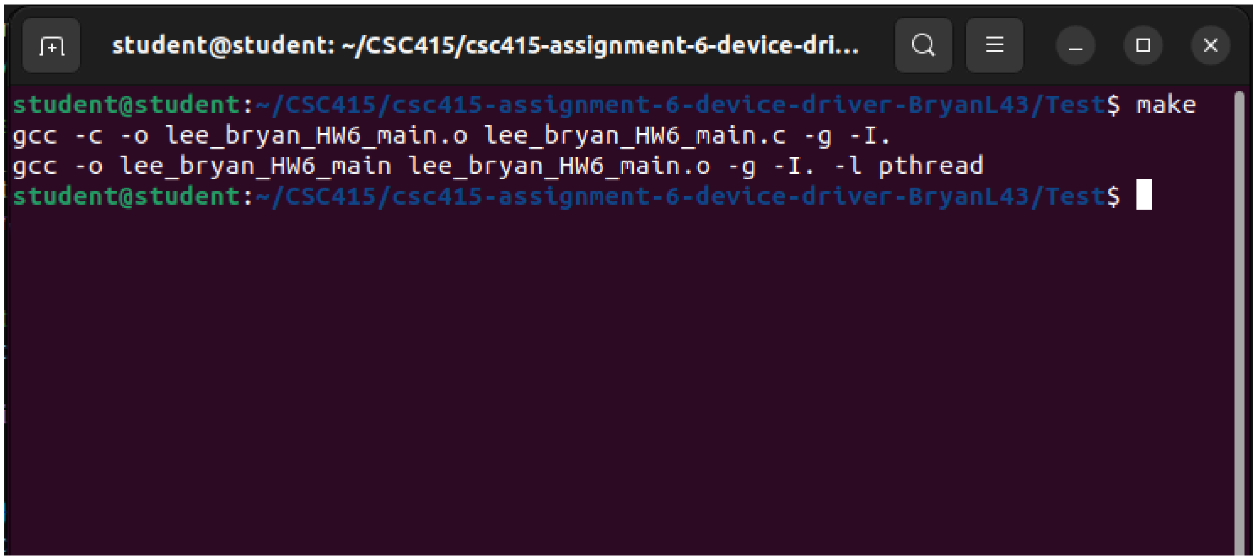
Screenshot of compilation:

Module:



```
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Module$ make
make -C /lib/modules/`uname -r`/build M=/home/student/CSC415/csc415-assignment-6
-device-driver-BryanL43/Module modules
make[1]: Entering directory '/usr/src/linux-headers-6.5.0-44-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04
) 12.3.0
  You are using:           gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  CC [M]  /home/student/CSC415/csc415-assignment-6-device-driver-BryanL43/Module
/Vigenere.o
  MODPOST /home/student/CSC415/csc415-assignment-6-device-driver-BryanL43/Module
/Module.symvers
  CC [M]  /home/student/CSC415/csc415-assignment-6-device-driver-BryanL43/Module
/Vigenere.mod.o
  LD [M]  /home/student/CSC415/csc415-assignment-6-device-driver-BryanL43/Module
/Vigenere.ko
  BTF [M] /home/student/CSC415/csc415-assignment-6-device-driver-BryanL43/Module
/Vigenere.ko
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-44-generic'
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Module$ ./in
stall.sh
[sudo] password for student:
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Module$
```

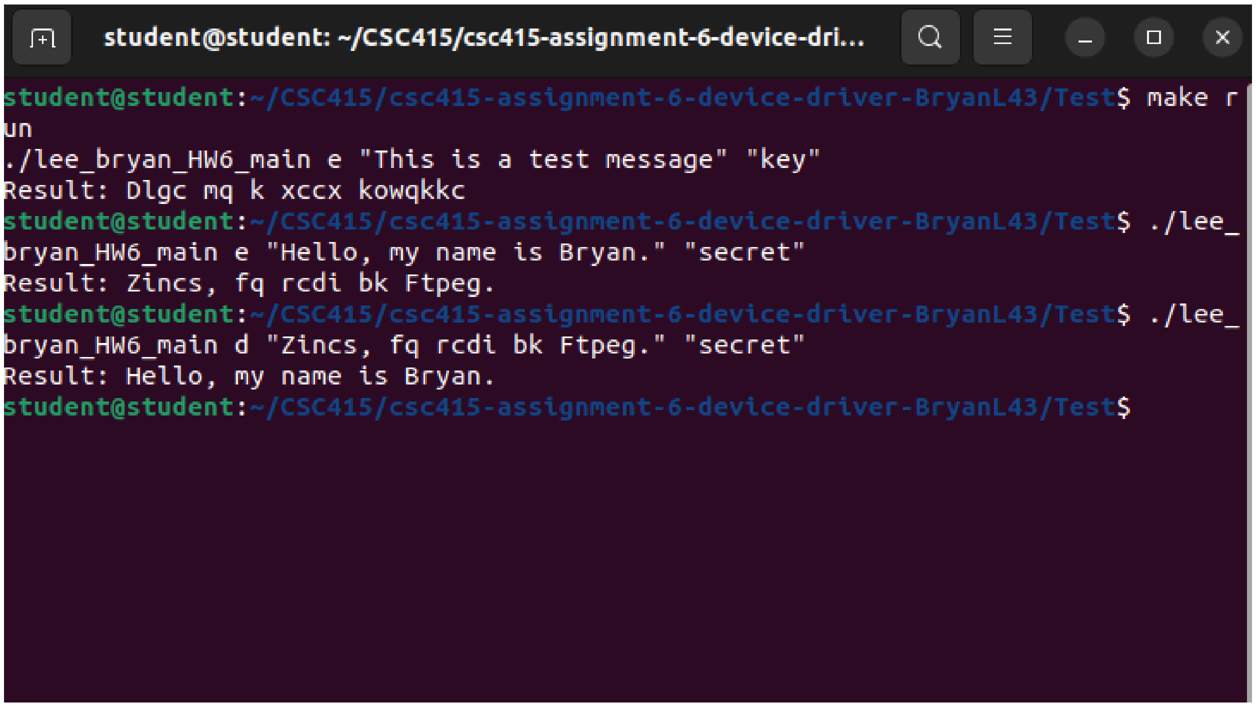
Test:



student@student: ~/CSC415/csc415-assignment-6-device-dri...

```
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$ make
gcc -c -o lee_bryan_HW6_main.o lee_bryan_HW6_main.c -g -I.
gcc -o lee_bryan_HW6_main lee_bryan_HW6_main.o -g -I. -l pthread
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$ 
```

Screenshot(s) of the execution of the program:



student@student: ~/CSC415/csc415-assignment-6-device-dri...

```
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$ make r
un
./lee_bryan_HW6_main e "This is a test message" "key"
Result: Dlgc mq k xccx kowqkkc
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$ ./lee_
bryan_HW6_main e "Hello, my name is Bryan." "secret"
Result: Zincs, fq rcdi bk Ftpeg.
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$ ./lee_
bryan_HW6_main d "Zincs, fq rcdi bk Ftpeg." "secret"
Result: Hello, my name is Bryan.
student@student:~/CSC415/csc415-assignment-6-device-driver-BryanL43/Test$
```