Design

-ioctl()

To prepare to use the ioctl() calls to hack into the kernel we had to use some resources from online. We first created a cryptctl device that would have all the ioctl() system calls defined to create a device, destroy a device, set key to devices, set size of key, get key and get size of key. Every device can have its own ioctl() commands, which can be an ioctl() to read (used to send information from a process to a kernel) and write (to return information to a process). The ioctl() system call that we used is:

```
used with kernel 2.6.35 or newer
and it is prototyped in <sys/ioctl.h>

long ioctl(struct file *f, unsigned int cmd, unsigned long arg);
```

The ioctl() function is called with three parameters: the file descriptor of the appropriate device file, the "cmd" - ioctl number, and a "arg" - parameter, which is of type long so you can use a cast to use it to pass anything. The "arg" can take in a structure or any they of arguments that could be passed into the char() device. The kernel can then read from the "arg" to the kernel space or write to the "arg" and return the information to the user space. The ioctl() number is usually created by a marco call `(_IO, _IOR, _IOW or IOWR, etc..)` The header file should be included both in the cryptctl.c and my_ioctl.c (app) to create, destory, etc devices.

-Encrypt/Decrypt Drivers

To prepare for this project, we read the three chapters of the Kernel Driver Book provided teaching the basics of char drivers. Our implementation of these subdrivers is therefore heavily based off of the source code provided by the book. Although the book's code is much more complex than other examples found online, the code book's code provides structures to handle large amounts of data, and it also handles synchronization. These can be seen be examining our data type Scull_Dev:

```
struct scull_qset{
    void **data;
    struct scull_qset *next;
};

/*this is my scull_struct*/
typedef struct scull_dev {
    struct scull_qset *data; /*pointer to first quantum set*/
    struct scull_dev *next;    /*pointer to the next listitem*/
    int quantum;            /*the current quantum size*/
    int qset;               /*the current array size*/
    unsigned long size;      /*amount of data stored here*/
    struct semaphore sem;    /*for mutual exclusion*/
    struct cdev cdev;        /*Char Device Structure*/
}Scull_Dev;
```

Each Scull_Dev is a linked list of pointers to other Scull_Dev structs. Each struct can hold at most 4 million bytes through an array of pointers (q_set->data). Each memory area is called a QUANTUM (defined to be 4000 bytes). Because we define the array to be 1000 indices, 1000 x 4000 = 4 million. This structure also has a semaphore to prevent mutual exclusion. We use this semaphore in the read() and write() calls to make sure multiple processes cannot modify the driver at the same time.

We decided to statically create only 6 device pairs. For the purpose of the project this number is arbitrary; we only wanted to show that the cryptctl device can handle more than 2 subdrivers. We also created one global variable key. This means that before encrypting or decrypting another device you

must change the key with the ioctl app explained below. This single global key can be used to show that the encryption/decryption is only successful if the proper key is used.

```
/*global variables for the encryption key*/
int key_size=0;
char *key;

/*start with major number = 0 (it will change in init()*/
static int simple_major = 0;
static int dsimple_major = 0;

/*start with minor number = 0 (it will increment with each creation) */
int scull_minor=0;


/*we have a static array of cdevs*/
static Scull_Dev HelloDevs[6];
static Scull_Dev dHelloDevs[6];
```

HelloDevs represents encrypting devices and dHelloDevs represents our decrypting devices. Our strategy was to first get the encryption working. We created the decryption by mimicking the encryption implementation so anything labeled with a ‘d’ before the variable is associated with the decryption devices. We also store the major of the subdrivers in global variables. These are dynamically allocated upon the creation of the first subdriver pair and are required when referencing devices through the MKDEV macro.

These two subdrivers have the same requirements but need different file_operations structures.

```
/*struct of our file operations for hello*/
static struct file_operations hello_fops = {
    .owner = THIS_MODULE,
    .open = hello_open,
    .release = hello_release,
    .read = hello_read,
    .write = hello_write,

};

/*struct of our file operations for decrypt*/
static struct file_operations dhello_fops = {
    .owner = THIS_MODULE,
    .open = hello_open,
    .release = hello_release,
    .read = hello_read,
    .write = decrypt_write,

};
```

This is because the write method must perform different actions. While the setup and preceding steps in both hello_write() and decrypt_write() are similar, each performs a different algorithm for encrypting and decrypting respectively. Here is the code for encryption within hello_write:

```
for(i=0;i<count-1;i++){
    if(message[i]>=97 && message[i]<=122){
        m = message[i]-97;
        k = key[j] - 97;
        n = (m+k) % 26;
        message[i]=(char)n+97;
        j++;
        if(j==key_size-1)
            j=0;
        }
}
```

The ecrpytion works based on the Vigenere cipher. The decryption is very similar except it needs to handle negative numbers.

```
for(i=0;i<count-1;i++){
    if(message[i]>=97 && message[i]<=122){

    /*get the values of letters*/
    m = message[i]-97;
    k = key[j] - 97;
    n = (m-k);
    /*if the difference is negative handle the cipher appropriately*/
    if(n<0){
        n=26+n;
    }
    else{
        n = n % 26;
    }
    message[i]=(char)n+97;
    j++;
    if(j==key_size-1)
        j=0;
    }
}
```

Difficulties:
The hardest part of this project was learning how to debug errors. It took a long time to interpret the Oops messages and understand where and what was causing the crash. There was also a large learning curve for understanding and developing a Char Driver. Creating the subdrivers to properly open/close took the longest. We also had slight difficulties with the cipher. We realized the when decrypting we had to handle negative errors.

-IOCTL APP

Before using anything we assume the user is root.
After calling insmod on our device (cryptctl.ko) the user call ioctl functions through our app.

1. First the user should call ./app -c to create a driver pair.
2. The user should next call ./app -ks <key> to set the cryptctl key.
3. The user can confirm the key any time by calling ./app -kg
4. Next the user should call ./app -w <message> <device> to write a message to the cryptEncrypt device. This will encrypt the message
5. The user can confirm the message is encrypted by calling ./app -r <device> to see the encrypted message
6. The user should copy the results from the previous call and then use ./app -w <copied_encrypted_msg> <device> to write the encrypted message to a cryptDecrypt device
7. The user can then confirm the message was decrypted by calling ./app -r <device>
8. The user can delete the pair by calling ./app –d
   Our deletion works like a stack. You can only delete/pop the pair most recently created

References:

http://linux.die.net/lkmpg/x892.html - used for ioctl() reference

http://www.linuxforu.com/2011/08/io-control-in-linux/ - used for ioctl() reference

http://lwn.net/Kernel/LDD3/ BOOK