**Calvin Bojanowski**

**CSC 4420 Final Project**

**Introduction**

As computing has advanced, we have seen many different platforms for storage such as hard disks, flash drive, solid state drives, etc. The newest form of storage that has become popular among businesses and average computer users is cloud storage. Cloud storage is a form of remote storage with a transaction between providers and customers.

Providers host servers and manage data stored on them. They then provide their storage to customers who store their data on the servers remotely. Often times providers will provide a fee for customers depending on how much data customers would like to store on provider’s servers.

Cloud storage comes with many useful features that make it more convenient to customers than physical storage. One of the key advantages being that, data stored on the cloud can be accessed from anywhere with internet access. Among some of the largest providers of cloud storage today (2019) are AWS (Amazon Web Services), AZURE, Google Cloud, IBM Cloud, and Oracle Cloud. In this report, I will be presenting my encryption/decryption for AWS.

AWS, also known as Amazon Web Services, is a subsidiary of Amazon that offers a cloud computing platform for individuals and customers. AWS is currently provides services to around 32% of cloud computing customers. Large corporations such as Netflix, Kellogg’s, and GE Oil & Gas use AWS to store their data.

Despite AWS and cloud storage’s rise, one concern among consumers has been the security of their data. Today you often hear about concerns over user data and how that data can be sold for money. Without proper security, user data such as credit cards, bank information, social security and other private information is at risk to be leaked, taken , or sold when stored remotely.

The purpose of this report is to help the reader replicate what I have done to add a layer of security when uploading files to Amazon Web Service’s s3 storage service. I provide the reader with the tools necessary to easily replicate or even build upon my method of encrypting and decrypting files uploaded to s3

**Project Goals**

The goal of this project project is to implement encryption and decryption into the open source code of s3fs. Normally, a user create a bucket in s3, mount that bucket to a local drive and put files into that local drive. From there, the files will upload to the remote bucket and a user can view them in the browser. In this project, the files will be caught before upload and encrypted. Users will also be able to upload encrypted files through the browser. Upon download, the files will be decrypted and users will be able to see the unencrypted file on their local drive. This project will use the RC4 encryption algorithm to encrypt and decrypt files. RC4 is used by Openssl, and our integrated encryption should be compatible with the Openssl encryption and decryption commands.

**System Information**

Below is a list of my system specs as well as software specs with a description when necessary:

**CPU INFO:**

**id:** cpu

**description:** CPU

**product:** Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

**vendor:** Intel Corp.

**physical id:** 4

**bus info:** cpu@0

**version:** Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

**serial:** To Be Filled By O.E.M.

**slot:** CPU 1

**size:** 905MHz

**capacity:** 3800MHz

**width:** 64 bits

**clock:** 400MHz

**capabilities:** x86-64 fpu fpu\_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp constant\_tsc arch\_perfmon pebs bts rep\_good nopl xtopology nonstop\_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds\_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4\_1 sse4\_2 x2apic popcnt tsc\_deadline\_timer xsave avx lahf\_lm epb tpr\_shadow vnmi flexpriority ept vpid xsaveopt dtherm arat pln pts cpufreq

**configuration:**

cores = 2

enabledcores = 1

threads = 2

**Memory Size:**

6 GiB

**OS/Kernal distribution:**

**Linux Mint (4.8.0-53-generic #56~16.04.1-Ubuntu SMP Tue May 16 01:18:56 UTC 2017 x86\_64 x86\_64 x86\_64 GNU/Linux): Linux Mint** is a community-driven [Linux distribution](https://en.wikipedia.org/wiki/Linux_distribution) based on [Debian](https://en.wikipedia.org/wiki/Debian) and [Ubuntu](https://en.wikipedia.org/wiki/Ubuntu_(operating_system)) that strives to be a "modern, elegant and comfortable [operating system](https://en.wikipedia.org/wiki/Operating_system)which is both powerful and [easy to use](https://en.wikipedia.org/wiki/Usability)."[[6]](https://en.wikipedia.org/wiki/Linux_Mint#cite_note-6) Linux Mint provides full [out-of-the-box](https://en.wikipedia.org/wiki/Out_of_the_box_(feature)) multimedia support by including some [proprietary software](https://en.wikipedia.org/wiki/Proprietary_software) and comes [bundled](https://en.wikipedia.org/wiki/Pre-installed_software) with a variety of [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source_software) applications.

The project was conceived by Clément Lefèbvre and is being actively developed by the Linux Mint Team and community.

(source: <https://en.wikipedia.org/wiki/Linux_Mint>)

download:<https://linuxmint.com/download.php>

**Packages & Tools:**

**FUSE:** FUSE (Filesystem in Userspace) is an interface for userspace programs to export a filesystem to the Linux kernel. The FUSE project consists of two components: the fuse kernel module (maintained in the regular kernel repositories) and the libfuse userspace library (maintained in this repository). libfuse provides the reference implementation for communicating with the FUSE kernel module.

A FUSE file system is typically implemented as a standalone application that links with libfuse. libfuse provides functions to mount the file system, unmount it, read requests from the kernel, and send responses back. libfuse offers two APIs: a "high-level", synchronous API, and a "low-level" asynchronous API. In both cases, incoming requests from the kernel are passed to the main program using callbacks. When using the high-level API, the callbacks may work with file names and paths instead of inodes, and processing of a request finishes when the callback function returns. When using the low-level API, the callbacks must work with inodes and responses must be sent explicitly using a separate set of API functions.

(source/download: <https://github.com/libfuse/libfuse>)

**s3fs v1.84:** s3fs is a FUSE filesystem that allows you to mount an Amazon S3 bucket as a local filesystem. It stores files natively and transparently in S3 (i.e., you can use other programs to access the same files). The maximum size of objects that s3fs can handle depends on Amazon S3. For example, up to 5 GB when using single PUT API. And up to 5 TB is supported when Multipart Upload API is used.

s3fs is stable and is being used in number of production environments, e.g., rsync backup to s3.

(source/download: <https://github.com/s3fs-fuse/s3fs-fuse>)

**OpenSSL:** OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library. For more information about the team and community around the project, or to start making your own contributions, start with the community page. To get the latest news, download the source, and so on, please see the sidebar or the buttons at the top of every page.

(source/download: <https://www.openssl.org/>)

**RC4 (part of OpenSSL):**  RC4 is a stream cipher and variable length key algorithm. This algorithm encrypts one byte at a time (or larger units on a time).  
A key input is pseudorandom bit generator that produces a stream 8-bit number that is unpredictable without knowledge of input key, The output of the generator is called key-stream, is combined one byte at a time with the plaintext stream cipher using X-OR operation.

(source: <https://www.geeksforgeeks.org/computer-network-rc4-encryption-algorithm/>)

**Vim (installed with Linux):** Vim is a highly configurable text editor for efficiently creating and changing any kind of text. It is included as "vi" with most UNIX systems and with Apple OS X.

(source: <https://www.vim.org/>)

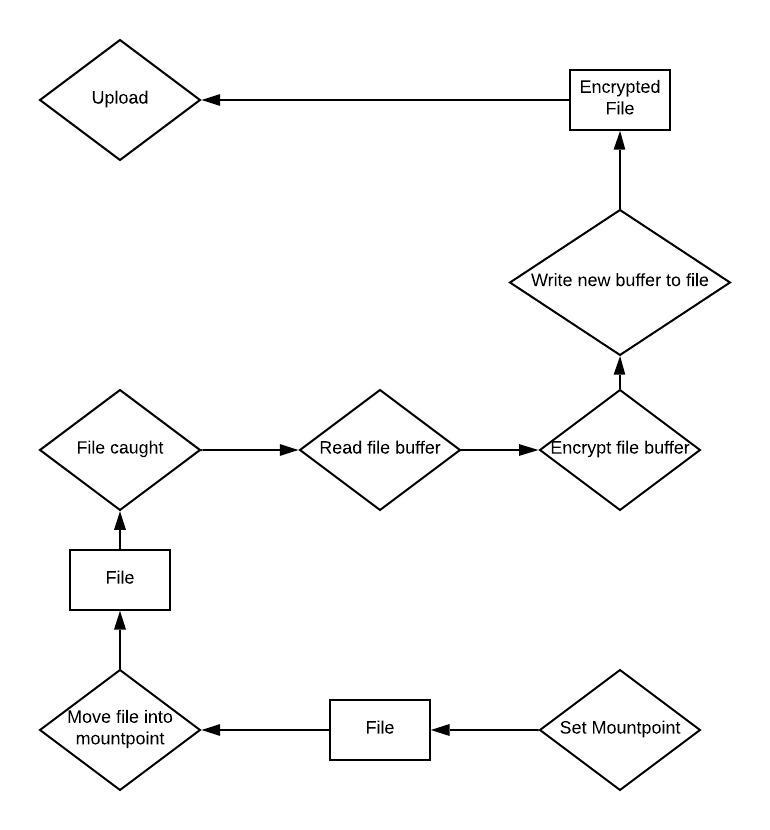
**GDB(installed with linux):** GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

(source: <https://www.gnu.org/software/gdb/>)

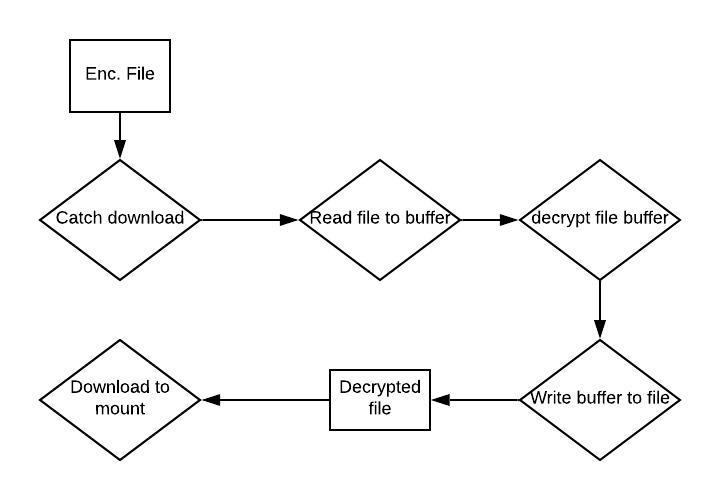
**GREP (installed with linux):** grep is a command-line utility for searching plain-text data sets for lines that match a regular expression. Its name comes from the ed command g/re/p (globally search a regular expression and print), which has the same effect: doing a global search with the regular expression and printing all matching lines. Grep was originally developed for the Unix operating system, but later available for all Unix-like systems and some others such as OS-9.

(source: <https://en.wikipedia.org/wiki/Grep>)

**DESIGN:**

Pictured below is a diagram of how my program is to work when uploading from mount point.

Here is the diagram for a download:



My first step in implementing my design, was to to find where s3fs calls its main function. Once I had found the file that contains ‘main()’ I could then search for functions related to uploading and downloading. Using a GREP search, I found the main function within a file named s3fs.cpp. While in s3fs.cpp, I searched for terms ‘upload’, ‘download’, ‘read’, and ‘write’. From here I began placing statements within the code that will print to a file called ‘log.txt’. From here I ran ‘make’ from the src directory so that I could create a new s3fs executable. When I ran this s3fs executable, I collected a log of what functions are called within s3fs.cpp. From here I went back in s3fs.cpp to analyze functions s3fs\_read and s3fs\_write. In these functions I found a variable ent of type FdEntity\* calling several different read and write functions. After discovering this, I decided to GREP search the directory for the declaration of FdEntity. I ended up discovering a file named fdcache.cpp within the src directory.

In my investigation of fdcache.cpp, I used the same search terms that I used while searching s3fs.cpp. Here I found functions Read, Write, Rowflush, and Load. As I did in s3fs.cpp, I put commands to log when these functions are called. One thing I found about the s3fs mount, is that functions are not just called when the s3fs program is executed. This meant that the log would be written to whenever I moved files in and out of the mount point.

As I moved files into my mount point, I analyzed the results printed to my log. Here the functions from Write and RowFlush from fdcache.cpp are called when I place a file into the mountpoint. Next,I uploaded a file from the browser and opened it. According to my log, functions Read, Load, and RowFlush are called.

Since I did not want my files to be encrypting and decrypting each time I was to read and write to a file, I would have to call my encrytion inside of Load and RowFlush. In this case RowFlush catches the upload while Load catches the download. At this point I began to design my encryption stand alone.

I started my stand alone encryption by analyzing the RC4 libraries in OpenSSL and payed specific attention to examples that were provided to us of how RC4 encryption is used. The main struggle of this part was figuring out what values to put into the RC4 functions in a way that is simple and in the context of a stand alone program. What I did know is that I needed to read a files contents into a buffer, get a passkey from the user, and create a new file containing the encrypted buffer.

After completing my standalone, with nosalt, I integrated it into the s3fs code. At that point I was able to demo my nosalt integration and standalone. After the demo, I continued to implement my salt in the standalone. I read as much documentation online about EVP\_BytestoKey() function as I could. In addition to that, I continued to GREP search the OpenSSL libraries for where salt is used. I first made an encryption algorithm that adds 16 bytes “Salted\_\_12345678” to the beginning of the encrypted file. After finding that this was compatible with Openssl I started my salt decryption in a separate program. For nosalt decryption, I first skipped the first 8 bytes, and read the next 8 into a buffer. Which was then read into EVP\_BytestoKey(), in addition to the rest of the encrypted file read into a buffer. My last and final step to creating my standalone was putting all of my encryption programs into one.

Of course I will go into the specific details of what my integration and standalone are capable of later in this document in addition to where in the s3fs code the integration took place.

**Integration:**

Here I will be discussing what parts of this project I have completed so far. The first part I will be talking about is my integrations with s3fs. My integration with s3fs is based on a stand alone RC4 encryption that I created to match OpenSSL’s encryption and decryption. One key aspect of my integration that I should note, is that it is not compatible with salt.

However, my rc4 standalone is much more powerful than the integrated encryption. I created a standalone function that is fully compatible with OpenSSL, in addition to being able to choose between salt/nosalt encryption and decryption. The user just has to specify what they want when theyrun the program. Using format ./rc4\_call <input\_file> <key> <output\_file> <Encrypt/Decrypt> <salt/nosalt>. The user can also use form ./rc4\_call <input\_file> <key> <output\_file> <Encrypt/Decrypt> if they want no salt.

If a user were to download an encrypted file in their bucket from the browser, they could run it through the standalone with no salt to decrypt it.

The only aspect that I believe went wrong was how much time I spent working to make sure my program was OpenSSL compatible. Only to find that the version on my system (1.0.2g) was not up to date with the program I was writing.

**Implementation Details**

Here I will be showing you the details of my s3fs integration. I will start by briefly showing how I kept track of what functions were called in s3fs.cpp.

Starting at line 4954, I created cout statements and wrote to the file log.txt.

**int main(int argc, char\* argv[])**

**{**

**/\*cout << "s3fs main function\n";**

**ofstream test;**

**test.open("//home//socks//Documents//4420\_project//log.txt",ofstream::out | ofstream::app);**

**test << "s3fs.cpp main" << endl;**

**test.close();//calvin\*/**

**int ch;**

**int fuse\_res; ...**

I have commented my logging out but this is how I first started tracking what functions were called. I also used this method to log which functions were called in fdcache.cpp.

In fdcache.cpp, here is the encryption function I created within the code, staring from Line 72:

**void calEncrypt(int fd){//calvin encryption**

**size\_t buffer\_size = lseek(fd, 0, SEEK\_END);**

**if (buffer\_size < 0) {**

**perror("no buffer size");**

**exit(0);**

**}**

**unsigned char input[buffer\_size];**

**unsigned char \*output = (unsigned char\*)malloc(buffer\_size);**

**lseek(fd, 0, SEEK\_SET);**

**if (read(fd, &input, buffer\_size) == -1) {**

**perror("file cannot be read");**

**exit(0);**

**}**

**char char\_key[] = "pass";**

**unsigned char generated[16];//for evp**

**RC4\_KEY key;**

**if (EVP\_BytesToKey(EVP\_rc4(),EVP\_sha256(),NULL,(const unsigned char\*)char\_key,strlen(char\_key),1,generated,NULL) < 0){**

**perror("Bytes to key no work");**

**exit(0);**

**};**

**RC4\_set\_key(&key, sizeof generated , (const unsigned char \*)generated);**

**RC4(&key, buffer\_size, (const unsigned char\*) input, output);//encrypt**

**/\***

**RC4\_set\_key(&key,strlen(char\_key), (const unsigned char \*)char\_key);**

**RC4(&key, buffer\_size, (const unsigned char\*) input, output);//encrypt**

**\*/**

**lseek(fd, 0, SEEK\_SET);**

**if (write(fd, output, buffer\_size) == -1) {**

**perror("file cannot be written to");**

**exit(0);**

**}**

**free(output);**

**}**

This function calEncrypt() takes the file data fd and gets the buffer size. After that, I read fd into array input and key “pass” into char\_key. Then I create an empty 16 byte array named generated, and pass char\_key, input, and generated into EVP\_BytesToKey(). Now that we have a generated key, I read generated into RC4\_set\_key(). I then put the key, input and output into RC4(). Output contains the encrypted data. I then sent the file pointer back to the beginning of fd so I can then write the output to fd.

I then called my function calEncrypt here at Line 1531:

**int FdEntity::RowFlush(const char\* tpath, bool force\_sync)**

**{**

**//calvin**

**/\* ofstream test;**

**test.open("//home//socks//Documents//4420\_project//log.txt",ofstream::out | ofstream::app);**

**test << "entering FdEntity::RowFlush\n";//calvin bojanowski**

**test.close();//calvin\*/**

**int result = 0;**

**S3FS\_PRN\_INFO3("[tpath=%s][path=%s][fd=%d]", SAFESTRPTR(tpath), path.c\_str(), fd);**

**if(-1 == fd){**

**return -EBADF;**

**}**

**AutoLock auto\_lock(&fdent\_lock);**

**calEncrypt(fd);//calvinencrypt**

**if(!force\_sync && !is\_modify){**

**// nothing to update.**

**return 0;**

**}**

**// If there is no loading all of the area, loading all area.**

**size\_t restsize = pagelist.GetTotalUnloadedPageSize();**

**if(0 < restsize){**

**if(0 == upload\_id.length()){**

**// check disk space**

**if(ReserveDiskSpace(restsize)){ ...**

And here at line 1277:

**int FdEntity::Load(off\_t start, size\_t size)**

**{ ...**

**calEncrypt(fd);//calvinencrypt**

**// Set loaded flag**

**pagelist.SetPageLoadedStatus((\*iter)->offset, static\_cast<off\_t>((\*iter)->bytes), true);**

**}**

**PageList::FreeList(unloaded\_list);**

**}**

**return result;**

**}**

As stated before, RowFlush is called when files are uploading and Load is called upon download.

Here I will discuss some important points about my stand-alone rc4 program.

Starting at line 45 of rc4\_Cal.cpp:

**unsigned char salted[8]= {'S','a','l','t','e','d','\_','\_'};**

**unsigned int salt\_array[8]={1,2,3,4,5,6,7,8};**

**if ( EVP\_BytesToKey(EVP\_rc4(),EVP\_sha256(),(const unsigned char \*)salt\_array,(const unsigned char \*)char\_key,strlen(char\_key),1,generated,NULL) < 0){**

**perror("Bytes to key no work");**

**exit(0);**

**};**

The highlighted portion is the salt that was generated. In this case I entered “12345678”. The next part I want to highlight is the part where I wrote the my salt to the file, starting at line 60:

**lseek(fd, 0, SEEK\_SET);**

**if (write(new\_file, salted, 8) == -1) {**

**perror("file cannot be written to");**

**exit(0);**

**}**

**if (write(new\_file, salt\_array, 8) == -1) {**

**perror("file cannot be written to");**

**exit(0);**

**}**

**if (write(new\_file, output, buffer\_size) == -1) {**

**perror("file cannot be written to");**

**exit(0);**

**}**

In this part I wrote “Salted\_\_”, “salt\_array”, and the rest of the encrypted output to the new\_file.

Now I would like to highlight some code from my salt decryption starting at line 85:

**unsigned char salted\_array[8];**

**unsigned char salt\_array[8];**

**if(read(fd,&salted\_array,8)== -1){**

**perror("file cannot be read");**

**exit(0);**

**}//read Salted\_\_**

**// printf("%s \n", salted\_array);**

**if(read(fd,&salt\_array,8)== -1){**

**perror("file cannot be read");**

**exit(0);**

**}//read salt**

**// printf("%s \n", salt\_array);**

**size\_t buffer\_size = lseek(fd, 0, SEEK\_END);**

**buffer\_size = buffer\_size - 16;**

**if (buffer\_size < 0) {**

**perror("no buffer size");**

**exit(0);**

**}**

Here I do the opposite of the salt encryption before I call EVP\_BytetoKey(). First I extract “Salted\_\_”, the salt, and the encrypted file data into their own arrays. After this I call EVP\_BytetoKey() the same way I did in the encryption.

**Future Improvements**

There are many improvements I could make to this project. The first thing that comes to mind is part of integration. In the integration, I would like users to create their own passkey when they mount the bucket to a local drive.

This process should not be too difficult. I already know how to write to a file on my machine. All it would require is writing to a file containing the key after ./s3fs is run. And then when it is time to encrypt and decrypt, that file is opened and read by my encryption function in fdcache.cpp.

Another Improvement I want to make with my integration is salt compatibility. And even further than that, I would like to make my integration work similar to my standalone, in which users can choose between salt and nosalt. This would require the s3fs program to remember whether or not the files are salted or not. My approach to implementing this would be similar to saving a passkey. With the difference being that the text file would indicate whether salt is true or false, somewhat like a configuration. The s3fs program would have to read the “configuration” file each time it is time to encrypt and decrypt to determine whether or not to call the salt or nosalt encryption.

My standalone encryption also has room for improvement. One major factor that could be improved is accounting for user error. Currently users will be thrown an error if they add too little arguments. However I have not added an error that will appear if too many arguments are called. Also, I want the program to run even if no arguments are entered. This would mean that my encryption would run its own shell and prompt the user for the rest of the arguments.

One final improvement I would like to make is in my integration with s3fs. A major hurdle in my project was getting my encryption to work with OpenSSL version 1.0.2g. Only to find that the rc4 documentation I was reading wasn’t compatible with this older version. My improvement would be a version of calEncrypt() that works with older versions of OpenSSL. This would mean reading through older OpenSSL documentation (which might be hard to find considering how the current documentation is somewhat poor) and creating the calEncrypt function based on that.

**Summary**

This project has given me a new insight on the software development process. What I really enjoyed was how hands on this assignment was, and how we were able to work with a piece of software that is tremendous in its popularity and power. My number one interest going into computer science was encryption and security and for a while I was lost not knowing the best way to learn these topics given the knowledge in other WSU CSC courses. I charpened my ability to read source code. In addition to reading code I properly learned to search for terms in the linux terminal. This project has given me a starting point for how to utilize encryption software in my own programs and in open source software. I feel as though if we did not do this project, I would have lost interest in encryption and security completely.