CSC 4420 Final Project Report

Jonathan Nabors, fj2262

1 INTRODUCTION

As the use of cloud-based storage increases, the cost of cloud-based storage solutions decreases. More and more technological giants such as Apple, Amazon, Microsoft, and Google are throwing their hat into the ring to provide a cloud-based storage system or a cloud-computing solution to take your data and hold it in their servers.

With individuals and enterprises alike both putting so much of their trust and stock in a cloud-based storage solution, they should equally be concerned with the security of their data. Since user's data is stored outside of the user's reach in some data warehouse somewhere, it is possible that their data can be stolen or manipulated in some way without the user ever knowing that this is happening or has happened.

By implementing some form of file encryption and decryption, cloud-based storage is granted a new and extra layer of security to keep precious and private data out of the hands of those who should not be interacting with this data to begin with.

We have seen multiple occurrences over the years of data leaks and hacks of cloud networks. How much more secure could these cloud-based storage services be if all of the data in the files were encrypted when stored on their storage drives? How much less severe would it be if systems were hacked and files stolen, but the hacker or thief could not read or interact with the data?

This report presents the design and implementation of file encryption and decryption at various stages of file upload and download to a cloud-based storage system. The goals of this project are stated below as well as the tools used and the outcome of the project.

2 PROJECT GOALS

The aim of this project was to successfully implement AES-CBC (Advanced Encryption Standard – Cipher Block Chaining) File Encryption & Decryption using Amazon's S3 cloud storage. When successfully implemented, whenever a file is created, edited, or placed into the local mount location, the file would be immediately encrypted and uploaded to the bucket storage location on Amazon s3.

All files in the local file system are to be automatically encrypted and the user can operate and use these files without explicitly having to call for any kind of decryption.

From the web interface of the Amazon S3 bucket, the file would remain encrypted. This means that if anyone were to find their way to the online bucket or repository, the contents of the data would still be encrypted and the hacker would not be able to decrypt the file without the secret key stored locally on the intended user's machine.

Whenever a file is moved from the online bucket to the local folder, the file in question would immediately become decrypted using a local encryption/decryption key stored somewhere on the user's computer (/home/jonny/Documents/MySecretKey.txt in this project's case). This key is a plain text string that is sixteen characters of text long and is retrieved and read at file encryption and decryption.

3 PACKAGES AND TOOLS USED

This project utilized a collection of existing software packages, frameworks, and libraries to achieve the intended goal. In some cases, these tools and packages were left in their original state and were not modified, however this was not always the case.

In the list below, each framework, software, or library will be listed as well as its intended purpose and use in the project. In Section 6 of this report, the modified source code will be presented in cases where additional code needed to be added or where existing source code was changed or altered.

1. Amazon Simple Storage Service (S3) – Online storage system for storing files and folder in the cloud. Amazon S3 is a secure, cloud-based web service interface that acts as a repository or bucket to hold a user's data. For this project, this is the web portal and back-end for data storage and warehousing. No changes were made to any existing Amazon source code or data files. More information about Amazon's S3 services can be found here: http://aws.amazon.com/s3/

2. s3fs – A FUSE-based file system backed by Amazon S3. A user is able to mount a bucket as a local file system for read/write actions. All files and folders are stored locally and are sent to and from the Amazon S3 bucket. A number of changes were made to the existing source code in this project and as such will be outlined in Section 6 of this report. More information about the s3fs project can be found here: https://code.google.com/p/s3fs/

3. FUSE – Provides implementation of a fully-functional file system in a user space. FUSE is what gives us the local file system that s3fs sits on top of and interacts with. The most commonly used feature of FUSE in this project is the mounting and un-mounting of the file system on the local machine to ensure connection and disconnection to the Amazon S3 bucket. No source code from the FUSE project was modified in any way during this project. More information about the FUSE project can be found here: http://fuse.sourceforge.net/

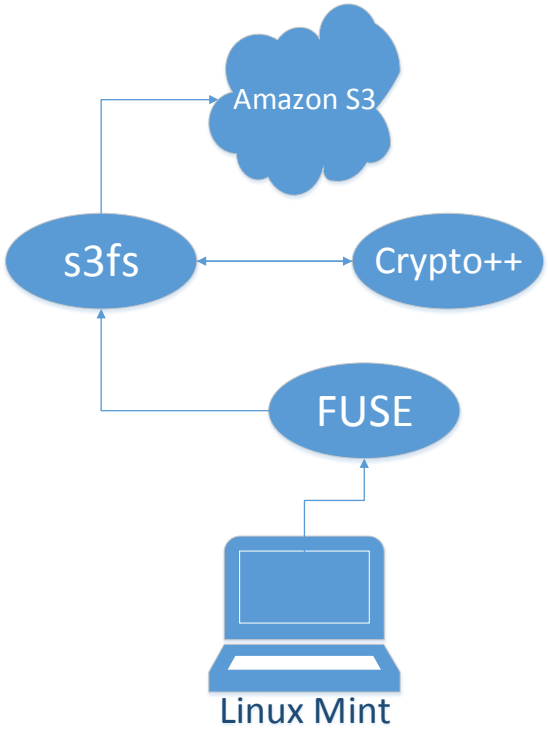
4. Crypto++ - A free C++ class library of cryptographic schemes containing algorithms used for the AES-CBC encryption/decryption used in this project. In addition to encrypting and decrypting text and files, Crypto++ also handles the random number generation used in this project. The Crypto++ classes are called throughout the modified s3fs source code to provide the advertised file encryption/decryption. No modifications were made to any existing Crypto++ library source code, though the Crypto++ header files are called in multiple places in the s3fs source code. More information about the Crypto++ Class Library can be found here: http://www.cryptopp.com/

5. Linux Mint – Mint is a Linux distribution that is compatible with Ubuntu. It provides an out of the box Linux solution that is based on Debian and Ubuntu. It is a reliable and safe operating system with little to no maintenance required. The entire project was researched, designed, developed, and implementing using Linux Mint as the sole operating system. More information about Linux Mint can be found here: http://www.linuxmint.com/

6. C++ - An object-oriented programming language used widely across the world. C++ is the programming language of choice for this project since it is what is already being used for s3fs as well as being the programming language used by Crypto++ (hence the 'plus-plus' at the end of Crypto++). Any additional code written in the header or source files for this project are written using C++ as the only programming language. More information about C++ can be found here: http://www.cplusplus.com/

4 System Information

This project was designed, debugged, and created on a computer running Linux Mint 17.1 and utilized s3fs version 1.74. The machine had 20.0 gigabytes of storage and 1.5 gigabytes of memory. The diagram below outlines the desired flow of the system from the user's machine all the way to the Amazon S3 cloud. We can visualize how data is moved from the user's file-system to the one provided by FUSE, then to its encrypted state by utilizing s3fs and Crypto++.



5 Design Considerations

During the design phase, the most important parts to understand of the project was where to actually implement the encryption and decryption in the pre-existing s3fs source code. Additionally, I would need to first implement a standalone file encryption and decryption process using Crypto++ as to become familiar with what the process flow would be like.

Using a provided template for AES-CBS encryption/decryption from Crypto++ (http://www.cryptopp.com/wiki/Advanced_Encryption_Standard) I was able to modify this class file to create an executable function that would encrypt and decrypt files for me using a secret key. After a number of tests using both binary and text files alike, the next step in the process was to find where to place these functions to allow the encryption and decryption to work in the flow of data from the file-system to the Amazon S3 cloud.

Deciding not to clutter up any current header and class files with my custom functions, I went the route of making my own separate class and header files for my custom encryption and decryption code. This was not a solution that worked out of the box, so there had to be some additional lines of code to be written into the Makefile.am and Makefile.in file in the source (src) folder of s3fs. After adding my required lines of code, I was able to run the installation command from the make files and properly link my class and header files in the compile instructions.

Placing the encryption and decryption calls in their functions came from a bit of trial and error. Knowing that we can't have decryption without proper encryption, the first task was to place the encryption function in each likely place until a proper location was found. Now, this does not mean that the encryption function found its way into each and every function and I just started marking them off one by one.

To find where to place the encryption function was the first step of the process. I began by looking into the s3fs.cpp source file data for a logical position to place the encryption function. My first search was a logical find request for the keyword 'write', thinking that I would be able to override an existing write command for replacing the file contents. I was able to begin my testing in the s3fs_write() function in the s3fs.cpp source file, however I was unable to implement my functionality in this location fully.

I moved away from the s3fs.cpp source file and into the fdcache.cpp file after doing more research. I placed my encryption function in its new location in the FdEntity::Write() function in the fdcache.cpp file at the very last line before the return statement. I was able to successfully encrypt the files with my function call here, but each time I encrypted I was greeted with a 'bad address' error message.

Not able to find out how to properly address my 'bad address' error message that was being called twice per file upload, I decided I wanted to find my third and final resting place for the encryption function call. I felt that my flaw in placing my encryption in FdEntity::Write() was that I was attempting to intercept the file contents using the provided 'bytes' variable. After retrieving the contents of the bytes variable I was then trying to change them in real-time so that I would not have to use any read or write methods. Though this may have been possible, I moved away from this method after not being able to successfully truncate the file contents with larger files.

At the recommendation of my TA and reading the email sent from the professor regarding possible locations, I settled on the FdEntity::RowFlush() function in the fdcache.cpp file for my file encryption. I placed my encryption function in its final resting place at line 883 of the fdcache.cpp source file, directly above the comment block that outlines how to handle the upload of files. I was pointed to this location while searching for the word 'upload' during my first look at the FdEntity::RowFlush() function. My goal for this was to just pass the file descriptor to my function to rewrite the contents of the file before it is uploaded and this was the best location I was able to find to achieve this goal.

After the encryption function has been called and completed its task of encrypting the files, the next step is to integrate the decryption. Following the same pattern as above of positional placement for the function, I landed into the FdEntity::Load() function for my placement of the decryption function.

I was able to settle on the FdEntity::Load() function after doing a test of where the functions are being called in the s3fs lifecycle. For this test, I wrote a method to write to the console whenever a specific function is being called. For my purposes, I was able to see that the FdEntity::Load() function was called at the proper time for me to call the decryption function.

The actual placement of the DecryptFile() function comes after the download operation has been performed on the file. A simple search for the keyword 'download' in the FdEntity::Load() function gave me a good indication of where to put the function call.

Finally, I chose to call the DecryptFile() function again in the FdEntity::RowFlush() method after the encryption has been done successfully. This is because the file is to remain decrypted in the local folder while encrypted on the cloud. The thought process behind this was that once the encrypted file has been placed on the cloud, we should then call the decryption function on it so that it remains in its unencrypted state on the local machine.

6 INTEGRATION

For this project, the goals were met for achieving both file encryption and decryption in the FUSE file system using s3fs and Amazon S3. Utilizing my own custom class and header files I have been able to encrypt and decrypt both binary and plain text files using a number of images to test the binary and text files for plain text encryption and decryption.

The way that the encryption works is that the file is first read and all of the contents, binary or plain text, are stored into a character array during the file upload portion of the fdcach::RowFlush() function. This intercepts the file before it is uploaded to the Amazon S3 bucket in the cloud and modifies and rewrites the content.

First, an IV is randomly generated and Hex Encoded into a readable string. This is what I believe introduces the issue of the file size doubling as it will take a string of 16 bytes and will increase it to 32 characters of readable text while Hex Encoded.

Secondly, the secret key is retrieved by calling the custom ReturnHiddenKey function which will read the contents of the file location that is passed as the first argument and will output it as a string .The secret key string is then Hex Decoded so it is back into its byte form instead of plain text. From here, the secret key is type casted into bytes form so that it may be used in the encryption.

After the secret key has been Hex Decoded and casted to byte form, it is used in conjunction with the IV to set the Encryption key. This encryption key is crucial in protecting the contents of the data and ensuring they cannot be read without the secret key and IV.

Next, the file contents are read and stored into a character array. This array is then passed to the StreamTransformationFilter for padding and allowing the encryption key to be set on the currently stored file contents. This is outputted as and considered to be the cipher text.

The cipher contents that have now been created are lastly Hex Encoded which will also have an effect on the size of the file, roughly doubling the size of the file contents similarly to how the IV is doubled in size during Hex Encoding.

Lastly, the encrypted IV and encrypted cipher are concatenated into a string. This string is moved to the buffer and its contents are written to the file instead of the unencrypted contents. This covers the end to end lifecycle of the encryption portion of the project. All of the files that are dropped into the mounted bucket will have their contents, either binary or in plaintext, encrypted via this method.

Continuing on to the decryption, which is performed during file load, we will see a number of the methods above in a slightly modified or reversed fashion. The benefit of the method in which I chose to implement the decryption is that it will preview and show the decrypted contents at all time in the mounted local bucket, but will always show the encoded contents online in the web interface for the Amazon S3 bucket.

To begin with the decryption, the program will first retrieve the secret key and will output the contents of the string. Similar to above, the secret key will be Hex Decoded into its byte form from its plain text version.

After the key has been retrieved, the file contents will again be read in the same way they are during encryption, but will be parsed using a different method. Since in encryption we are creating the IV and then adding the cipher text to the end, we need to give the decryption method a way to read first the IV, and then read the cipher afterwards and store them in variables so that we can utilize them later in the decryption.

To accomplish the separation of the IV and cipher text, we utilize the substring method from the C++ library (.substr()). We separate the IV from the file contents first by reading the first 32 characters of the string starting from the beginning of the file. Since the IV has been Hex Encoded, it is 32 characters long instead of 16 characters long in its non-Hex-Encoded state hence the requirement to read only the first 32 characters.

The cipher context is read similarly by reading from where the IV contents end and until the end of the file. The length of the file contents are gathered while the file is being read at the beginning of the decryption function call.

Next, the file contents are ran through a Hex Decoder to change them from plain text to byte form by casting the StringSink of the Hex Decoder into a byte array. This allows us to declare and set our Decryption key with the IV and secret key in their byte forms.

The only final step remaining is to actually decrypt the contents of the file with the decryption key and IV that are set in the Decryption variable above. The Hex Decoded file contents are ran through the StreamTransformationFilter with the decryption key and are stored into a string for the final operation.

Lastly, like in the encryption method, the file contents of the decrypted file are rewritten with the newly decrypted values created in this function.

7 IMPLEMENTATION DETAILS

As state in the above sections, all modified pre-existing source code is related to the following files:

-fdcache.cpp

-/src/Makefile.in

-/src/Makefile.am

The supplemental function calls that I have written are in their own class and header files located in the /src/ folder of the s3fs folder. The file names are:

-fj2262encrypt.h

-fj2262encrypt.cpp

All of the code in the two above files handle the functions related to the secret key retrieval, encryption, and decryption of the files. The fj2262encrypt.cpp and fj2262encrypt.h files contain three functions, each one handling a different aspect of the project. The first function, ReturnHiddenKey(const char* nameOfFile, string secretKey) takes two arguments, the first being the location of the file passed as a string whose contents are the absolute path to the secret key file that must be read, and the second being the return value of the function. The sample call for this method would be:

String myKey;
ReturnHiddenKey("/home/jonny/Documents/mySecretKey.txt", myKey);

The next function is the EncryptFile(int fd) function which handles all of the file encryption. The execution of this method is covered above in section 6 and it only takes as its single argument the file descriptor for the file that needs to be encrypted.

Lastly, the third and final function covered is the DecryptFile(int fd) function, also covered above in section 6. Like the encryption function, this takes as its only parameter the file descriptor of the file that needs to have its contents decrypted.

The functions calls can be found in the modified fdcache.cpp file by either doing a search for the function names EncryptFile(fd) or DecryptFile(fd) or by searching for 'Jonny'. There are comments before the encryption and decryption functions are called in the FdEntity::RowFlush() and FdEntity::Load() functions.

Additionally, you may find the function calls specifically at lines 884, 927, and 824 of the fdcache.cpp source file. Also, the include statement for the fj2262encrypt.h file can be found on line 42 of the fdcache.cpp file. This include statement is necessary for calling the functions in the fj2262encrypt.cpp file and is necessary for the successful operation of this program.

8 FUTURE IMPROVEMENTS

The implementation of this program contains one outlying flaw that I would have hoped to have perfected if I had allowed myself more time. With the reading and storage of the file contents into their

respective variables during encryption, the contents of the file are ran through a Hex Encoder call using the Crypto++ functions. Moving something through the Hex Encoder will double the size of whatever is being read into something base 16. So if it is 16 bytes long and ran through a Hex Encoder, it will now instead be 32 characters long in plain text. The doubling of the file sizes isn't so much of a big deal for files that are only a few bytes large, but if you have files that are hundreds of megabytes or gigabytes and you are doubling their size then you will quickly run into performance and storage space issues in the application.

Going forward, I would have like to have removed the calls to Hex Encoding the contents of the file and just leaving them in the byte formats when applicable as to not force the size of the files to double during the encryption. Much of the lines of code involve encoded and then decoding the contents of whatever is being read or stored. I'm sure that with more time I would be able to find suitable replacements for these calls or would have been able to find places in the functions where extraneous code may be removed.

In addition to addressing the issue of file sizes increasing, I would also like to look for other suitable locations for the file encryption and decryption function calls. I am not convinced that the most optimal location for file encryption and decryption are in the FdEntity::Load() and FdEntity::RowFlush() functions.

I would have liked to have kept with the FdEntity::Read() and FdEntity::Write() functions as the home of the encryption and decryption methods, since I think that there was efficiency in rewriting the contents of the bytes variable, however in the intent to complete the project I left them in their current working place. To perfect the functions in the FdEntity::Read() and FdEntity::Write() methods would have taken an unknown amount of time to find out why the decryption specifically did not fully decrypt files or would only write parts of the decrypted file contents.

9 SUMMARY

The s3fs file encryption and decryption projects has been one of the most rewarding experiences of my undergraduate career. What started off as a simple concept was able to evolve into what at times felt like a behemoth of work that would never be finished. The key factors in being able to complete this project were driven by the timeline posted by the professor.

The biggest learning points that I have been able to take away from this project can be summed up in two categories. The first has been a huge amount of gained knowledge about the general workings of the Linux operating system and system calls. Before this project, I had very limited exposure to Linux operating systems and how to interact with them. I am now able to successfully use and navigate through my operating systems using the terminal alone, rarely having to look at an actual graphical user interface while on a Linux based machine. Much of this familiarity with the terminal has come from the many repeated calls to move, read, copy, or delete files while testing out the encryption and decryption function.

Second, and possibly more important, I have gained an immense amount of knowledge in the field of cryptography and AES-CBC encryption and decryption. This project was a rewarding exercise in cryptography for file systems and understanding at what point in the file movement lifecycle that one

would want to either encrypt or decrypt the contents of their file for reading. I feel that the knowledge gained from this aspect alone will serve me well throughout my job hunts and careers outside of my current university setting.