

Introduction

The following file includes details about each part of the assignment. The design decisions and reasons why the designs are secure overlap for each section. Due to this, these reasonings have been combined to avoid repetition. Reasoning for specific parts are annotated with the part they relate to.

Design Decisions

- Ensuring the correct arguments and number of arguments are entered

I used this design decision to ensure the user enters the correct arguments. This will reduce the amount of possible errors that could occur. The code won't and shouldn't work unless all the arguments are given hence, by checking arguments, this means no encryption / decryption code is run until the correct arguments are entered.

- Displaying information error messages which don't give away important information

I used this design decision to ensure that if an error occurs, the user can determine why the error occurred. However, the error messages which are shown won't reveal important information. This is important as it means there aren't any information leaks, making the code more secure.

- Using CipherOutputStream for encryption and CipherInputStream decryption

These have been used as these cipher streams are secure and allow for easy encryption and decryption of information.

- Generating random information such as salt, IV and keys using SecureRandom

This information needs to be randomly generated to ensure the code is more secure, an attacker is less likely to guess the values, and the values don't give away important information. Due to this, I have used SecureRandom as it is a "cryptographically strong random number generator" as per <https://www.baeldung.com/java-secure-random>. This ensures the randomly generated values are quite strong and random.

It also means that for part 2, even when encrypting the same file with the same password, the results are different as a different salt is used each time.

Furthermore, for parts 3 and 4, by randomly generating this information, it means that if a user uses the same password multiple times or encrypts the same file, the generated key will be different, as will the files.

- Only displaying the IV (for part 1) and secret key if the encryption is successful

I chose this design decision as the generated password and IV only need to be displayed if the encryption was successful, otherwise they are useless.

- Randomly generating 128 bit salt for **(Parts 2-4)**

I chose to randomly generate salts which were 128 bits in length in order to adhere to the NIST specifications. This is to make my design more secure as NIST states that, "[t]he salt SHALL be at least 32 bits in length and be chosen arbitrarily so as to minimize salt value collisions among stored hashes."¹

- Storing the IV at the start of the encrypted file **(PART 2)** and also the salt **(PART 3)**

¹ <https://pages.nist.gov/800-63-3/sp800-63b.html#sec5>

I used this design choice because firstly, the IV and salt don't need to be encrypted. Even if an attacker has this information, they still need the key before the file can be decrypted. Furthermore, everytime a file is encrypted, a random salt and IV is created. Due to this, this information is completely unrelated to the plaintext or the keys, hence they reveal no information about the plaintext or the key. This is another reason why they do not need to be a secret.

I also added the IV and salt to the start of the file, as this meant they are easy to read and access.

- Storing the algorithm, key length, IV and salt at the start of the encrypted file **(PART 4)**

I used this design choice because firstly, this information doesn't need to be encrypted. Even if an attacker has this information which has been used for the encryption of the file and creation of the key, they still need the password and/or the key before it can be decrypted. Furthermore, this information reveals nothing about the plaintext or password and so it doesn't need to be a secret. I also added this information to the start of the file, as this meant it was easy to read and access these values which is necessary for decrypting files and querying the metadata.

- Storing the algorithm and key length by also including how long each of them are **(PART 4)**

The information which is stored in the encrypted file, relating to storing the algorithm and key length, is described below:

- Store one byte representing the length of the algorithm in bytes
- Store the algorithm in bytes
- Store one byte representing the length of the key length in bytes
- Store the key length in bytes

I used this technique as the algorithm and key length aren't always going to be the same length and as such, the code is unable to read a fixed number to get the correct values. This method allows the program to easily and efficiently retrieve the algorithm and the key length. The program will read one byte which represents the length of the cipher and then use this, to read the correct number of bytes for the cipher. This will be repeated for the key length.

Why the design is secure:

Part 1:

My design is secure because I randomly generate the information which isn't given and I do this by using SecureRandom. I also ensure the error messages displayed don't reveal important information. Furthermore, CipherOutputStream and CipherInputStream are used as they represent a secure cipher stream.

Part 2:

Building on what was stated above, another reason why the design in part 2 is secure is because the random IV (which is used as a salt) means that encrypting the same file with the same key, produces different results. Furthermore, only the IV is stored in the encrypted file, and the IV doesn't give away any information about the plaintext or the key. Hence, making the design more secure. Another reason why the design is secure is because according to NIST, "[t]he salt SHALL be at least 32 bits in length and be chosen arbitrarily so as to minimize salt value collisions among stored hashes."² In my program, the IV (which is used as a salt) is 128 bits and it is randomly generated each time the program is run, hence adhering to these specifications.

Part 3 and 4:

² <https://pages.nist.gov/800-63-3/sp800-63b.html#sec5>

Building on what was stated above, the code uses a random salt and a high number of iterations to create the key from the password. This makes the key quite random and hard to guess. The salt which is used to generate the key from the password is also always random. This means that even when using the same password multiple times, the generated key will be different, as will the encrypted file. Furthermore, the information which is stored in the file does not give away information about the plaintext or the password. Hence, making the design more secure. Another reason why my design is secure is because it uses the “PBKDF2WithHmacSHA256” algorithm to generate the key and according to NIST, PBKDF2 is a suitable derivation function. The algorithm also uses HMAC which is an approved one-way function, which is required according to NIST. In addition to these, PBKDF2 uses an iteration count. This should typically be at least 10,000 iterations according to NIST and at least 310,000 iterations according to OWASP³. To ensure the program adheres to these requirements, a count of 400,000 is used, making the design secure.

Part 1:

How it works:

The program for part 1 works by determining whether the user wants to encrypt or decrypt (by looking at the arguments) and then calling a relevant method which performs the action.

For encryption, the program randomly generates a key and an IV and uses them, along with the cipher algorithm, to create and initialize the cipher. The relevant input and output files are then opened and the cipher is used to encrypt the data, and write it to the output file.

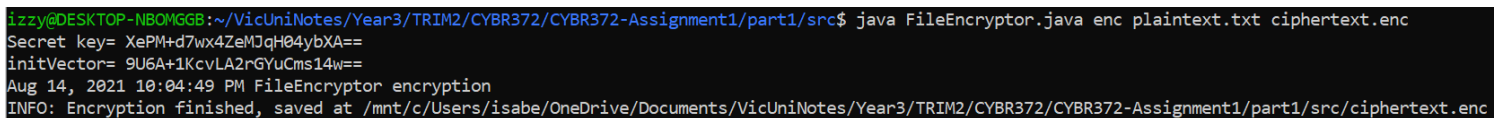
For decryption, the program first base64 decodes the key and IV (which the user gives the program) and uses this, along with the cipher algorithm, to create and initialize the cipher. The relevant input and output files are then opened and the cipher is used to decrypt the data, and write it to the output file.

Code meets the requirements:

Encryption:

When entering the following command, the file is encrypted successfully and the secret key and IV are outputted in base 64 encoding as shown from the screenshot below:

```
java FileEncryptor.java enc plaintext.txt ciphertext.enc
```



```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1/src$ java FileEncryptor.java enc plaintext.txt ciphertext.enc
Secret key= XePM+d7wx4ZeMJqH04ybXA==
initVector= 9U6A+1KcvLA2rGYuCms14w==
Aug 14, 2021 10:04:49 PM FileEncryptor encryption
INFO: Encryption finished, saved at /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1/src/ciphertext.enc
```

Decryption:

When decrypting the file, the following command is used, where the base 64 encoded secret key and IV are both specified along with the input and output files. This decrypts the file successfully as shown in the screenshot below:

```
java FileEncryptor.java dec XePM+d7wx4ZeMJqH04ybXA==
9U6A+1KcvLA2rGYuCms14w== ciphertext.enc plaintext1.txt
```

³ https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1/src$ java FileEncryptor.java dec XePM+d7wx4ZeMJqH04ybXA== 9U6A+1KcvLA2rGYu
Cms14w== ciphertxt.enc plaintext1.txt
Aug 14, 2021 10:06:31 PM FileEncryptor decryption
INFO: Decryption complete, open /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1/src/plaintext1.txt

```

To compare the original file (plaintext.txt) to the decrypted file (plaintext1.txt) hexdump is used to verify that they are the same. The following screenshot shows that the two files are the same and so the code successfully decrypted the file:

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1$ hexdump -b plaintext.txt
00000000 150 145 154 154 157 040 164 150 151 163 040 151 163 040 164 145
00000010 170 164 015 012 015 012 015 012 145 156 143 162 171 160 164 040
00000020 155 145 041 041 041
00000025
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1$ hexdump -b plaintext1.txt
00000000 150 145 154 154 157 040 164 150 151 163 040 151 163 040 164 145
00000010 170 164 015 012 015 012 015 012 145 156 143 162 171 160 164 040
00000020 155 145 041 041 041
00000025
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part1$

```

This shows that my code meets the requirements as it allows the user to specify encryption or decryption operations, the secret key and IV in base 64 encoding (for decryption) and the input and output file paths as per the requirements. The code also correctly encrypts and decrypts the files.

Part 2:

How it works:

The program for part 2 works by determining whether the user wants to encrypt or decrypt (by looking at the arguments) and then calling a relevant method which performs the action.

For encryption, the program decrypts the key given by the user and randomly generates an IV. These are used, along with the cipher algorithm, to create and initialize the cipher. The relevant input and output files are then opened to read from and write to respectively. The IV is written to the start of the output file, then the cipher is used to encrypt the data, and write it to the output file.

For decryption, the program first base64 decodes the key and then opens both the input and output files. The first 16 bytes of the file are read to retrieve the IV value, and this value is used, along with the key and cipher algorithm, to create the cipher. The cipher is then used to decrypt the data, and write it to the output file.

Code meets the requirements:

Encryption:

When entering the following command, the file is encrypted successfully as shown from the screenshot below. This command allows the user to specify the base 64 secret key as well as the input and output files.

```

java FileEncryptor.java enc ZLTVMx2Jk3KCXWTX6S9VRA== plaintext.txt
ciphertxt.enc

```

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2/src$ java FileEncryptor.java enc ZLTVMx2Jk3KCXWTX6S9VRA== plaintext.txt
ciphertxt.enc
Aug 14, 2021 10:11:42 PM FileEncryptor encryption
INFO: Encryption finished, saved at /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2/src/ciphertxt.enc

```

Decryption:

When decrypting the file, the following command is used, where the base 64 encoded secret key is specified along with the input and output files. This decrypts the file successfully as shown in the screenshot.

```
java FileEncryptor.java dec ZLTVMx2Jk3KCXWTX6S9VRA==  
ciphertext.enc plaintext1.txt
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2/src$ java FileEncryptor.java dec ZLTVMx2Jk3KCXWTX6S9VRA== ciphertext.enc plaintext1.txt  
Aug 14, 2021 10:12:31 PM FileEncryptor decryption  
INFO: Decryption complete, open /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2/src/plaintext1.txt
```

To compare the original file (plaintext.txt) to the decrypted file (plaintext1.txt) hexdump is used to verify that they are the same. The following screenshot shows that the two files are the same and so the code successfully decrypted the file:

```
README.md assignment1.iml ciphertext.enc ciphertext1.enc 0000 plaintext.txt plaintext1.txt 0000  
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$ hexdump -b plaintext.txt  
00000000 124 150 151 163 040 151 163 040 141 040 146 151 154 145 041 041  
00000010 015 012 015 012 015 012 120 154 145 141 163 145 040 145 156 143  
00000020 162 160 171 164 040 155 145 041  
00000028  
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$ hexdump -b plaintext1.txt  
00000000 124 150 151 163 040 151 163 040 141 040 146 151 154 145 041 041  
00000010 015 012 015 012 015 012 120 154 145 141 163 145 040 145 156 143  
00000020 162 160 171 164 040 155 145 041  
00000028  
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$
```

From this we can see that the code meets the requirements of part 2 as the secret key can be specified and the IV doesn't need to be specified during decryption. The code also correctly encrypts and decrypts the files.

Furthermore, when encrypting the exact same file using the same secret key multiple times, the output is always different. This is because the program randomly generates an IV (which is used as a salt) each time encryption occurs. This IV/salt is used in the encryption as explained above, and is the reason why the output is always different. I have shown this below:

The following two commands have been used to create two encrypted files, using the same input file and secret key.

```
java FileEncryptor.java enc ZLTVMx2Jk3KCXWTX6S9VRA== plaintext.txt  
ciphertext.enc
```

```
java FileEncryptor.java enc ZLTVMx2Jk3KCXWTX6S9VRA== plaintext.txt  
ciphertext1.enc
```

When using the hexdump tool on both output files we can see that its contents are very different as shown in the screenshot below:

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$ hexdump -b ciphertext.enc
00000000 136 026 201 211 302 003 263 265 057 060 330 350 236 205 150 225
00000010 123 256 044 152 145 251 016 010 264 270 377 004 214 065 317 217
00000020 242 002 165 160 153 104 363 263 073 132 216 304 366 001 267 350
00000030
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$ hexdump -b ciphertext1.enc
00000000 316 352 014 366 276 203 361 075 275 014 371 174 342 156 123 164
00000010 224 013 037 002 171 202 261 023 160 373 061 156 240 332 351 266
00000020 077 207 213 331 120 021 323 350 370 323 070 372 265 331 172 203
00000030
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part2$

```

This shows that the cipher text is different each time.

Part 3:

How it works:

The program for part 3 works by determining whether the user wants to encrypt or decrypt (by looking at the arguments) and then calling a relevant method which performs the action.

For encryption, the program randomly creates an IV and salt. The key is then generated from the given password, using the random salt, a fixed number of iterations and a key length. This key is then used, along with the IV and cipher algorithm, to create and initialize the cipher. The relevant input and output files are then opened to read from and write to respectively. The salt and IV are written to the start of the output file, and then the cipher is used to encrypt the data, and write it to the file.

For decryption, the program firstly opens both the input and output files. The input file is used to retrieve the salt and the IV by reading the first 16 bytes for each. The key is then generated from the given password, and the salt which has been retrieved. This key is then used, along with the IV and cipher algorithm, to create and initialize the cipher. The cipher is then used to decrypt the data, and write it to the output file.

Code meets the requirements:

Encryption:

When entering the following command, the file is encrypted successfully as shown from the screenshot below. This command allows the user to specify a password along with the input and output files. The created key, made from adding salt, iterating and hashing repeatedly is displayed to the user.

```
java FileEncryptor.java enc "my password" plaintext.txt
ciphertext.enc
```

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3/src$ java FileEncryptor.java enc "my password" plaintext.txt
ciphertext.enc
password= u2u3Ni8Z6NNDTbuvHXuBpg==
Aug 14, 2021 10:15:02 PM FileEncryptor encryption
INFO: Encryption finished, saved at /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3/src/ciphe
rtext.enc

```

Decryption:

When decrypting the file, the following command is used, where only the password, input and output files are specified. This decrypts the file successfully as shown in the screenshot.

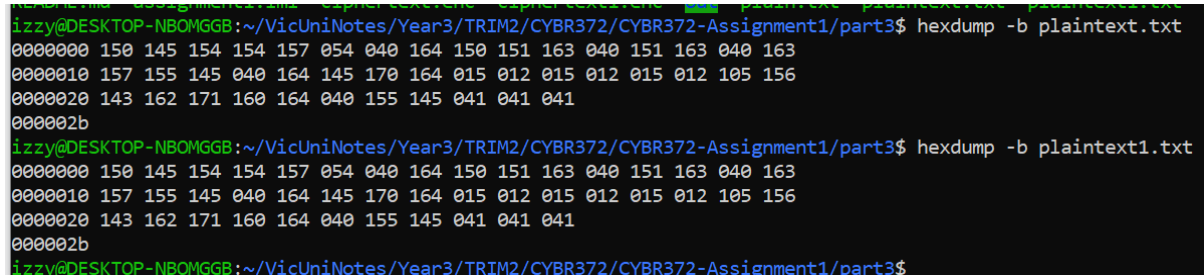
```
java FileEncryptor.java dec "my password" ciphertext.enc
plaintext1.txt
```

```

izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3/src$ java FileEncryptor.java dec "my password" ciphertext.enc
plaintext1.txt
Aug 14, 2021 10:15:38 PM FileEncryptor decryption
INFO: Decryption complete, open /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3/src/plaintext1
.txt

```

To compare the original file (plaintext.txt) to the decrypted file (plaintext1.txt) hexdump is used to verify that they are the same. The following screenshot shows that the two files are the same and so the code successfully decrypted the file:



```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3$ hexdump -b plaintext.txt
00000000 150 145 154 154 157 054 040 164 150 151 163 040 151 163 040 163
00000010 157 155 145 040 164 145 170 164 015 012 015 012 015 012 105 156
00000020 143 162 171 160 164 040 155 145 041 041 041
0000002b
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3$ hexdump -b plaintext1.txt
00000000 150 145 154 154 157 054 040 164 150 151 163 040 151 163 040 163
00000010 157 155 145 040 164 145 170 164 015 012 015 012 015 012 105 156
00000020 143 162 171 160 164 040 155 145 041 041 041
0000002b
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part3$
```

From this we can see that the code meets the requirements of part 3 as a secret key is generated from the password and the files are successfully encrypted and decrypted. The code also correctly encrypts and decrypts the files.

Part 4:

How it works:

The program for part 4 works by determining whether the user wants to encrypt, decrypt or get information about the file (by looking at the arguments). Then, the program calls the relevant method which performs the correct action.

For encryption, the program determines via arguments given, both the algorithm and the key length to be used. Then the program randomly creates an IV and salt, used for encryption and key generation. The length of the IV is set based on the algorithm, for example, AES has an IV length of 16 whereas Blowfish has an IV length of 8. The key is then generated from the given password using the specified algorithm, the random salt, a fixed number of iterations and the specified key length. This key gets used, along with the IV and cipher algorithm, to create and initialize the cipher. The relevant input and output files are then opened to read from and write to respectively. The algorithm, key length, salt and IV are written to the start of the output file and then the cipher is used to encrypt the data, and write it to the file. As the algorithm and key length aren't fixed lengths like the salt, the program writes to the file one byte which represents the length of the algorithm, and then it writes the algorithm in bytes. The same process is used for writing the key length to the file. This is explained in more detail in the design decisions section.

For decryption, the program firstly opens both the input and output files. The input file is used to retrieve the algorithm, key length, salt and the IV. The algorithm and key length are retrieved by reading one byte to determine how long the algorithm is, then reading that many bytes to retrieve the algorithm. This process is then repeated to retrieve the key length. The salt and IV are then retrieved by reading the first 16 bytes for the salt and either 16 or 8 bytes for the IV depending on the algorithm. The key is then generated from the given password, and the algorithm, key length and salt which have been retrieved from the encrypted file. This key is then used, along with the IV and cipher algorithm, to create and initialize the cipher. The cipher is then used to decrypt the data, and write it to the output file.

For information, the program reads the encrypted file, and uses the technique mentioned above to retrieve the algorithm and key length. These are then displayed to the user.

Code meets the requirements:

Encryption:

When entering the following command, the file is encrypted successfully as shown from the screenshot below. This command allows the user to specify an algorithm, the key length, and a password along with the input and output files. The created key, made from adding salt, iterating and hashing repeatedly is displayed to the user.

```
java FileEncryptor.java enc AES 128 "my password" plaintext.txt  
ciphertext.enc
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java enc AES 128 "my password" plaintext.txt ciphertext.enc  
password= utHq5i8F3KM1Cl5C+XObPw==  
Aug 14, 2021 10:17:08 PM FileEncryptor encryption  
INFO: Encryption finished, saved at /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src/ciphertext.enc
```

Decryption:

When decrypting the file, the following command is used, where only the password, input and output files are specified. This decrypts the file successfully as shown in the screenshot:

```
java FileEncryptor.java dec "my password" ciphertext.enc  
plaintext1.txt
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java dec "my password" ciphertext.enc plaintext1.txt  
Aug 14, 2021 10:17:25 PM FileEncryptor decryption  
INFO: Decryption complete, open /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src/plaintext1.txt
```

To compare the original file (plaintext.txt) to the decrypted file (plaintext1.txt) hexdump is used to verify that they are the same. The following screenshot shows that the two files are the same and so the code successfully decrypted the file:

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4$ hexdump -b plaintext.txt  
00000000 150 145 154 154 157 054 040 164 150 151 163 040 151 163 040 163  
00000010 157 155 145 040 164 145 170 164 015 012 015 012 120 154 145 141  
00000020 163 145 040 145 156 143 162 171 160 164 040 155 145 056 056 056  
00000030 015 012 015 012 104 165 156 040 144 165 156 040 144 165 156 015  
00000040 012 015 012 124 145 163 164 151 156 147 040 164 150 151 163 040  
00000050 146 151 154 145  
00000054  
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4$ hexdump -b plaintext1.txt  
00000000 150 145 154 154 157 054 040 164 150 151 163 040 151 163 040 163  
00000010 157 155 145 040 164 145 170 164 015 012 015 012 120 154 145 141  
00000020 163 145 040 145 156 143 162 171 160 164 040 155 145 056 056 056  
00000030 015 012 015 012 104 165 156 040 144 165 156 040 144 165 156 015  
00000040 012 015 012 124 145 163 164 151 156 147 040 164 150 151 163 040  
00000050 146 151 154 145  
00000054  
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4$
```

Extracting Metadata:

One of the requirements of part 4 is to "allow the user to query the metadata embedded in the file." By entering the following command, this can be done as shown in the screenshot below. The code reads the file to extract the data and displays it to the user.

```
java FileEncryptor.java info ciphertext.enc
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java info ciphertext.enc  
AES 128
```


The examples above show that the AES algorithm with a key length of 128 works. The following example shows that the Blowfish algorithm with a key length of 256 also works to verify that part 4 adheres to the requirements.

```
java FileEncryptor.java enc Blowfish 256 "my password"  
plaintext.txt ciphertext.enc
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java enc Blowfish 256 "my password"  
plaintext.txt ciphertext.enc  
password= BodUbccFdBjkDJ65iN+xmrWqntcRTAW3duaTrw9XHBI=  
Aug 14, 2021 10:18:12 PM FileEncryptor encryption  
INFO: Encryption finished, saved at /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src/ciphe  
rtext.enc
```

```
java FileEncryptor.java dec "my password" ciphertext.enc  
plaintext1.txt
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java dec "my password" ciphertext.enc  
plaintext1.txt  
Aug 14, 2021 10:18:26 PM FileEncryptor decryption  
INFO: Decryption complete, open /mnt/c/Users/isabe/OneDrive/Documents/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src/plaintext1  
.txt
```

```
java FileEncryptor.java info ciphertext.enc
```

```
izzy@DESKTOP-NBOMGGB:~/VicUniNotes/Year3/TRIM2/CYBR372/CYBR372-Assignment1/part4/src$ java FileEncryptor.java info ciphertext.enc  
Blowfish 256
```

From this we can see that the code meets the requirements of part 4 as the user can specify the algorithm, (either Blowfish or AES) as well as the key length. The algorithm and key length used to encrypt the file is also stored in the file, this data is used to decrypt the file and the user is able to query this information. The code also correctly encrypts and decrypts the files.