

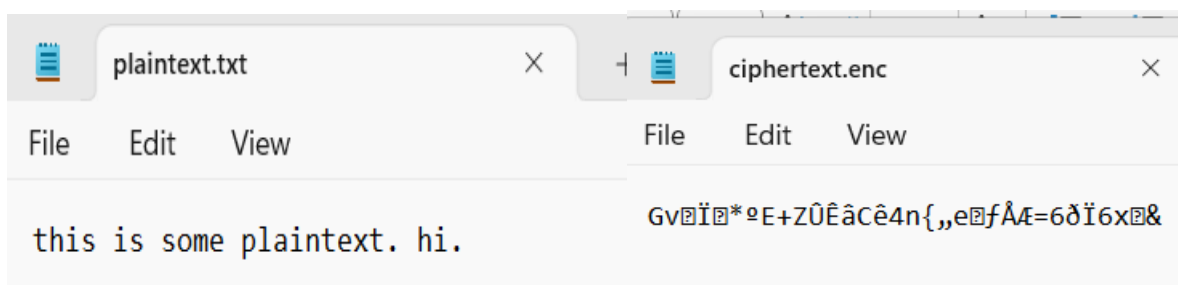
Cam Olssen  
300492582  
cam.olssen@gmail.com

#### Part 1:

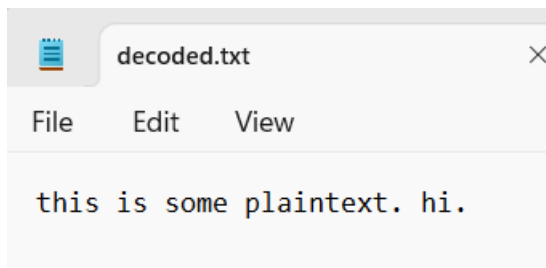
I added two new methods to the provided code – ‘enc’ and ‘dec’, which encrypt and decrypt respectively. The main method now creates a new FileEncryptor and runs the ‘run’ method, which runs the specified operation based on the arguments passed into the program. The required arguments are the operation (enc or dec), the input file and the output file.

The encryption method takes the key, initialization vector, input file path and output file path as parameters. It then largely just runs the example code, though modified to replace the hardcoded input and output with custom locations based on the arguments.

The decryption method takes the same parameters, using the key and IV to decrypt the file – again, largely the same as in the example code but with user-customisable inputs and outputs.



Plaintext input and ciphertext output of running part 1's encryption method.



Decoded output of the decryption method being run on the above ciphertext.enc.

#### Part 2:

I changed the arguments from the part 1 version, with both methods requiring 4 arguments and throwing an error otherwise. The decryption method no longer takes the IV as a parameter. The required arguments for both operations are the operation, the key, the input and the output.

The IV is generated when the program is run, and prepended to the encrypted file. When the decrypt method is run, it reads the first 16 bytes of the ciphertext file as the input vector, and uses that to decrypt the data, which is why it no longer takes the IV as a parameter.

The IV is not encrypted, as it does not pose any particular security threat to do so, as the same ‘salt’ will never be generated for the same key, and any attacker would need to have both the IV and secret key in order to decrypt a file.

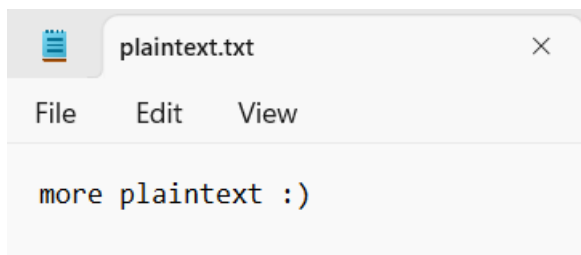
Ciphertext.enc and ciphertext2.enc both used the same key in their encryption (aOGfudhlLfmVzkCZNwZSjQ==), but we can see that use of the Hexdump for Windows utility (sourced from <https://www.di-mgt.com.au/hexdump-for-windows.html>) that both encryptions of the

Cam Olssen  
300492582  
cam.olssen@gmail.com

same file with the same key produced a different ciphertext result, showing that the IV is acting as a salt.

```
C:\Users\camol\OneDrive\Documents\uni\cybr372\assignment1\src\part2>hexdump ciphertext2.enc
00000000  af 75 24 31 16 42 73 d9 d1 1c 3f da 8f 6b bd 8f
00000100  a6 1b 7f 60 67 be 75 05 5e e8 33 6d 34 8c 5d 5e
00000200  20 32 15 3d 60 d5 2c 30 f0 7a 53 c5 da 4f 79 17

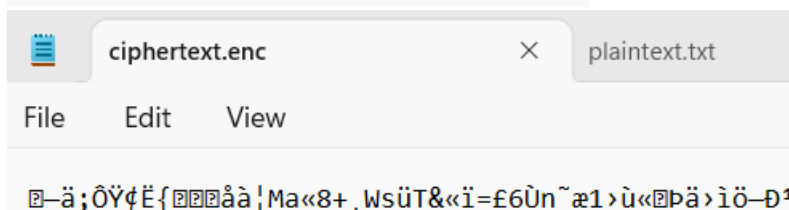
C:\Users\camol\OneDrive\Documents\uni\cybr372\assignment1\src\part2>hexdump ciphertext.enc
00000000  1b 97 e4 3b d4 9f a2 cb 7b 02 16 7f e5 e0 a6 4d
00000100  61 ab 38 2b b8 57 73 fc 54 26 ab ef 3d a3 36 d9
00000200  6e 98 e6 31 9b f9 ab 18 de e4 9b ec f6 97 d0 b9
```



plaintext.txt

File Edit View

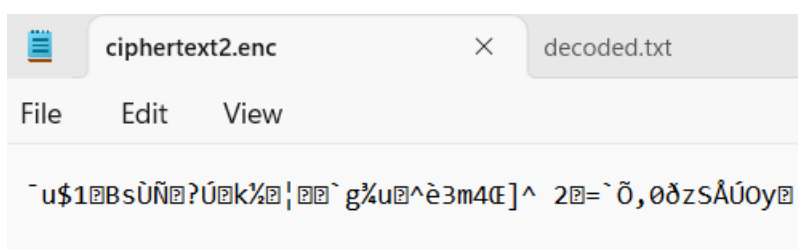
more plaintext :)



ciphertext.enc

File Edit View

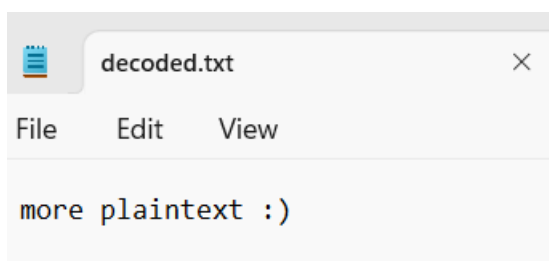
ä;ÖŸçË{åå|Ma«8+,WsüT&«ï=£6Ûn~æ1>ù«ä»ä>ïö-Ð¹



ciphertext2.enc

File Edit View

u\$1BsÜÑ?Úk%| g%u^è3m4E]^ 2= `Ö,øðzSÅÚOy



decoded.txt

File Edit View

more plaintext :)

As above, outcomes of encryption and decryption methods in Part 2. We can see that different ciphertext is generated for ciphertext.enc and ciphertext2.enc, despite their use of the same encryption key.

### Part 3:

The required arguments are now the operation, the password, input and output. The run method converts the second argument, the password, from a string into a char array. This is because if it were stored as a string, there would be no way to zero out the contents as strings are immutable.

Cam Olssen  
300492582  
cam.olssen@gmail.com

I added a random-generated 16-byte salt to the encryption method, and made use of the provided algorithms to salt, iterate and hash 1000 times in order to generate a 128-bit secret key based off the provided password. The salt is stored similarly to the IV in part 2, prepended to the encrypted file.

The decryption method now also takes the password char array as a parameter. It reads the salt and IV from the encrypted file, and uses them to re-generate the secret key based on the password provided, which is then used to decrypt the file.

```
C:\Users\camol\OneDrive\Documents\uni\cybr372\assignment1\src\part3>hexdump ciphertext.enc
000000 98 c9 52 70 d7 50 d7 58 2d a5 8c 2c 0e de b9 24
000010 22 ad 41 e0 45 a8 49 3a 68 ff 9a 8d e9 d5 6f e1
000020 3c 81 bf 16 d6 e3 5d d1 e2 3d 7f 88 1e a4 34 f5
000030 cf 3e e6 d8 e9 23 e3 8e 77 e2 2c ce 4e 98 18 d7

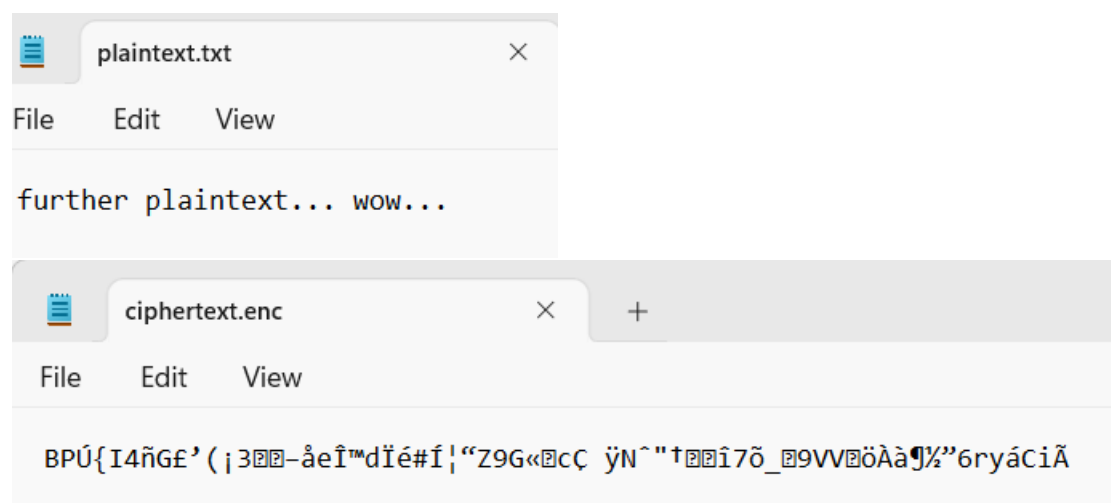
C:\Users\camol\OneDrive\Documents\uni\cybr372\assignment1\src\part3>hexdump ciphertext2.enc
000000 77 fa 00 f1 c8 59 04 32 5f e1 90 3a 19 55 c0 f8
000010 c5 f3 79 5c 9d 2b cb a3 4f 2d 80 e4 da 1c 68 51
000020 bf 7a ae 7f 2c c6 3b d0 fc 6f f3 50 9a 57 72 c5
000030 12 91 31 1f 75 f6 5e ae ba 37 b2 a1 1c 57 be 16
```

Based on the hexdump output we can observe that the same file, encrypted twice using the same password ("password"), will produce different results, showing that the salt and IV are working correctly to ensure security.

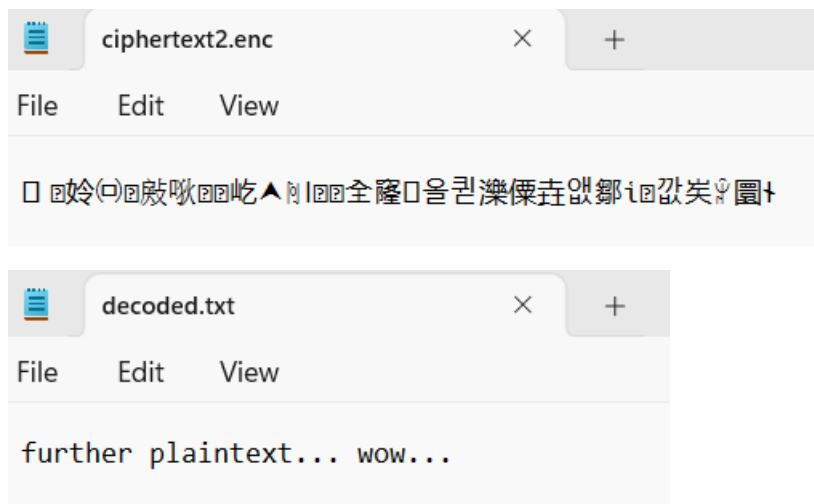
```
Secret key = mhqwm/j43yWLYUJ2LesC6Q==
Aug 17, 2023 9:13:47 PM com.packtpub.crypto.section5.src.part3.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc
```

```
Secret key = FIa9sCR0cD7BEU50mTEuWg==
Aug 17, 2023 9:11:08 PM com.packtpub.crypto.section5.src.part3.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext2.enc
```

The two encryptions, both using "password" as the password. This also shows that the key generated is unique to each encryption operation, even with the same password used.



Cam Olssen  
300492582  
cam.olssen@gmail.com



Outputs of encryption and decryption for Part 3. As previously, we can see that despite using the same password, ciphertext.enc and ciphertext2.enc contain different ciphertext.

#### Part 4:

The program can now take three arguments for the operation instead of just 'enc' or 'dec' – 'info' is now an option, which runs the newly added 'info' method. For encryption, arguments are now required for the algorithm and key length.

The encryption operation now takes additional arguments for the algorithm used (AES or Blowfish) and the length of the key generated. If the second argument is not "AES" or "Blowfish", the program will throw an exception due to an invalid algorithm. It will then check the key length to ensure that it fits with what is permitted by the selected algorithm – either 128, 192 or 256 for AES, and any number from 32 to 448 for Blowfish. It still takes a password as specified in Part 3.

The encryption method will write the algorithm and key length used to the metadata of the encrypted file using the UserDefinedFileAttributeView interface. A new method has been added called 'getFileAttribute', which takes the metadata of a file and outputs it as an ArrayList of strings.

The newly added 'info' method uses this to get and output the attributes of a file. The decryption method also uses getFileAttributes in order to learn the algorithm and key length, which it uses to decrypt the file.

```
Aug 17, 2023 9:24:25 PM com.packtpub.crypto.section5.src.part4.FileEncryptor run
INFO: Invalid number of arguments!
```

Attempting to run the program without adding the algorithm or key length.

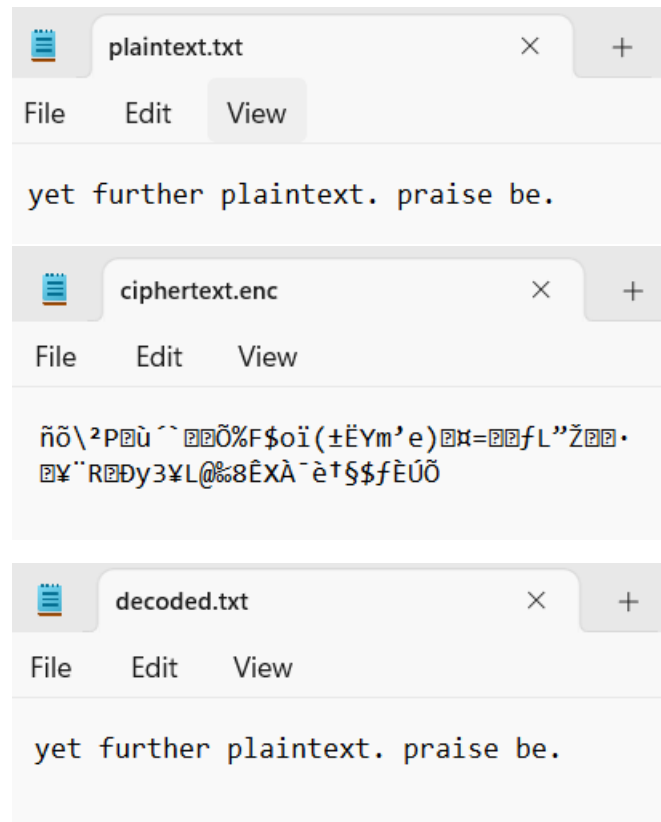
```
Secret key = ngThFP5IBU0ut7eEqIRorVn210Qxea1RxMd+jf3nd/w=
Algorithm = AES
Key length = 256
Aug 17, 2023 9:26:35 PM com.packtpub.crypto.section5.src.part4.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc
```

Output of a successful encryption.

## 256 AES

Output of the 'info' operation on the resulting ciphertext.enc.

The IV generation has been changed, where the size of the IV array is now based on the algorithm used, whereas previously it could be left as 16 bytes due to the defaulting to AES. In the event that the program manages to run without a specified algorithm or key length, which should be impossible, it will default to using AES with a 128-bit key, which the previous parts used.



As previously, output generated by encryption and decryption using Part 4's version of the FileEncryptor.

### Part 5:

As specified, I implemented the vPNG.java class for a vulnerable pseudorandom number generator. The constructor takes a specified long as a seed, and in the next() method I chose to use a simple linear congruential generator algorithm to update the seed and generate a pseudorandom value.

```
seed = (seed * 1103515245 + 12345) & 0x7FFFFFFF
```

This is the line which runs the LCG algorithm, using the equation below.

$$X_{n+1} = (aX_n + c)$$

Cam Olssen  
300492582  
cam.olssen@gmail.com

In short, the LCG algorithm takes the seed value  $X_n$ , multiplies it by the provided multiplier  $a$ , and adds the increment  $c$ . It also applies the bitmask  $0x7FFFFFFF$ , which is to ensure that the result generated remains within the range of positive 32-bit integers. This updated value becomes the seed for the new iteration, and the process is repeated to generate subsequent pseudorandom values.

In this case  $a = 1103515245$  and  $c = 12345$ . Both of these values were chosen completely arbitrarily by myself.

FileEncryptor.java has been updated to use this instead of the SecureRandom class to generate the IV, with a preset seed of "12345678", again a meaningless number chosen arbitrarily by myself. A custom seed can be entered after the initial arguments for encryption.

```
C:\Users\camol\.jdk\corretto-1.8.0_362\bin\java.exe ...
Seed: 12345678
Random values from vPNG: 6F 7C 05 5A 8B 68 81 26 67 14 BD B2 03 80 B9 FE
Random key=D4zhQQK2S0HEomKyfJyMbg==
initVector=b3wFWotogSZnFL2yA4C5/g==
Aug 18, 2023 1:47:32 PM com.packtpub.crypto.section5.src.part5.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc

C:\Users\camol\.jdk\corretto-1.8.0_362\bin\java.exe ...
Seed: 12345678
Random values from vPNG: 6F 7C 05 5A 8B 68 81 26 67 14 BD B2 03 80 B9 FE
Random key=j4r32x661kY0u/9301QsvQ==
initVector=b3wFWotogSZnFL2yA4C5/g==
Aug 18, 2023 1:49:04 PM com.packtpub.crypto.section5.src.part5.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext2.enc
```

Here we have the outputs from running FileEncryptor twice using the default seed. As we can see, the values generated from vPNG.java are the same every time if the same seed is used. Using a different seed will generate different pseudorandom values, as shown here.

```
Seed: 87654321
Random values from vPNG: 96 17 04 ED 22 B3 70 E9 6E 0F 9C A5 7A 2B 88 21
Random key=4BVRzvw9uwy7CroQzIyfIA==
initVector=lhcE7SKzc0luD5yleiuIIQ==
Aug 18, 2023 1:50:11 PM com.packtpub.crypto.section5.src.part5.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc
```

The random key is still different as it is not generated using vPNG but the SecureRandom class, but the initVector is the same when the same key is used, as is the generated random values, showing that vPNG.java is deterministic.

This is not a cryptographically secure generator. If an attacker were to know that vPNG.java was being used for random value generation, the fact that it will always generate the same values for the same seeds means that they could use it to reverse-engineer or guess the seed value and predict the sequence that would be output for the initialisation vector – allowing them to potentially begin decrypting the encryption algorithm used altogether. Though the secret key in this example is still secure, if vPNG was used to generate a key for as well, the vulnerabilities would be even more

Cam Olssen  
300492582  
cam.olssen@gmail.com

exacerbated, as the key from a certain seed being the same every time would make decryption far easier.

In order to make this generator less vulnerable, several things could be done. Use of a random seed rather than a fixed one is one, with it ideally being generated based on a secure entropy source such as keyboard timings, mouse movements and IDE timings, as is done by the Linux kernel. Changing the algorithm used for random value generation would also be advisable, as the LCG algorithm that we have used is a poor implementation.