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.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated  
Plaintext input and ciphertext output of running part 1’s encryption method.

A screenshot of a computer

Description automatically generated  
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 (aOGfudhlLfMVkzCZNwZSjQ==), 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 same file with the same key produced a different ciphertext result, showing that the IV is acting as a salt.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
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.

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.

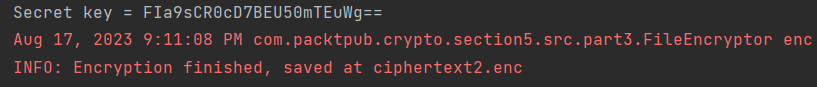
A computer screen with white text

Description automatically generated

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.

A screen shot of a computer

Description automatically generated



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.

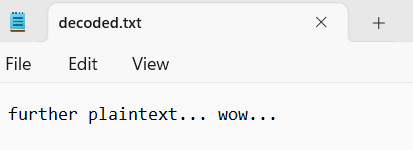
A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



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:

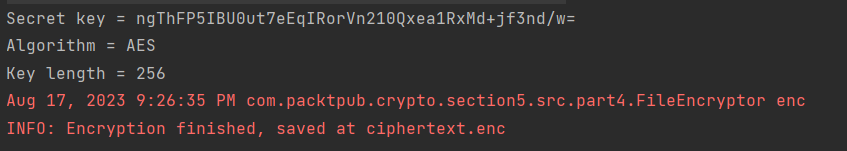
The program can now take three arguments for the operation instead of just ‘enc’ or ‘dec’ – ‘info’ is now an option, which runs the newly added ‘info’ method. For encryption, arguments are now required for the algorithm and key length.

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.

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



Output of a successful encryption.

A black background with white text

Description automatically generated

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.

A screenshot of a phone

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

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.



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

In short, the LCG algorithm takes the seed value *Xn*, multiplies it by the provided multiplier *a*, and adds the increment *c*. It also applies the bitmask 0x7FFFFFF, 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”. A custom seed can be entered after the initial arguments for encryption.