Part 1: Basic File Encryption and Decryption

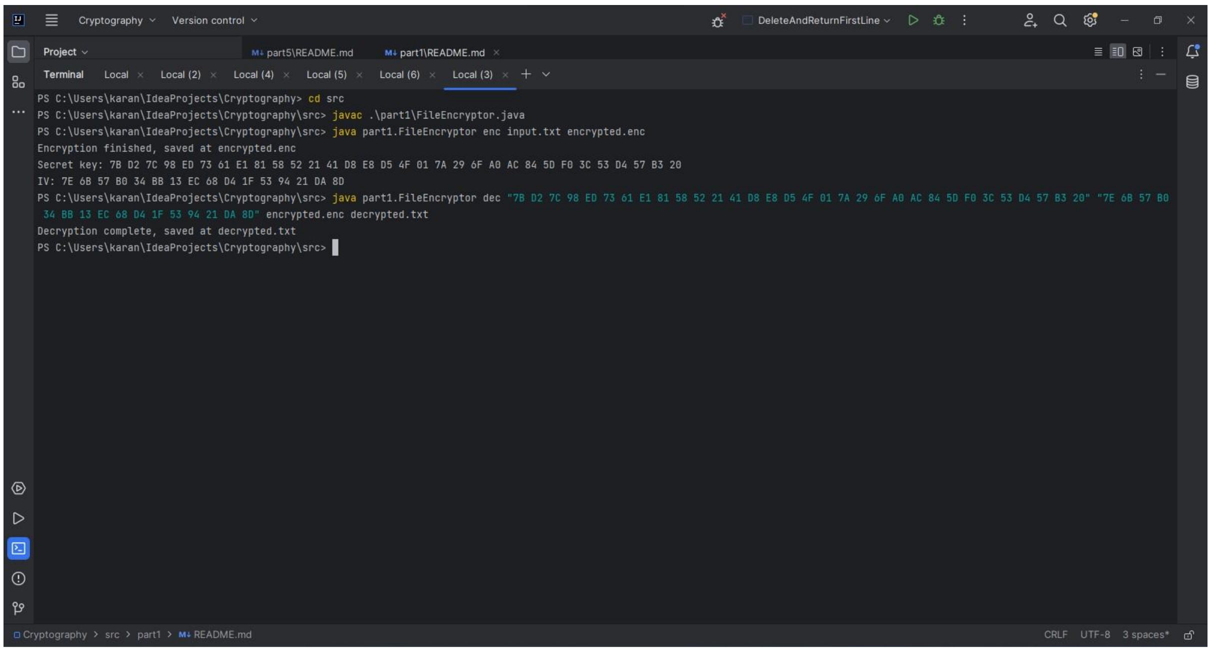
Introduction: In Part 1 of the assignment, I implemented a basic file encryption and decryption program using a symmetric key algorithm. The goal was to showcase the encryption and decryption process for a sample input file.

Implementation Steps:

1. Read the input file "input.txt."
2. Generate a secret key using a password-based approach.
3. Initialize the cipher with the secret key and AES algorithm.
4. Perform encryption on the input file and create "encrypted.enc."
5. Perform decryption on "encrypted.enc" to obtain the decrypted output.

Screenshots:

- Running encryption process



```
PS C:\Users\karan\IdeaProjects\Cryptography> cd src
PS C:\Users\karan\IdeaProjects\Cryptography\src> javac .\part1\FileEncryptor.java
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part1.FileEncryptor enc input.txt encrypted.enc
Encryption finished, saved at encrypted.enc
Secret key: 7B D2 7C 98 ED 73 61 E1 81 58 52 21 41 D8 E8 D5 4F 01 7A 29 6F A0 AC 84 5D F0 3C 53 D4 57 B3 20
IV: 7E 6B 57 B0 34 BB 13 EC 68 D4 1F 53 94 21 DA 8D
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part1.FileEncryptor dec "7B D2 7C 98 ED 73 61 E1 81 58 52 21 41 D8 E8 D5 4F 01 7A 29 6F A0 AC 84 5D F0 3C 53 D4 57 B3 20" "7E 6B 57 B0
 34 BB 13 EC 68 D4 1F 53 94 21 DA 8D" encrypted.enc decrypted.txt
Decryption complete, saved at decrypted.txt
PS C:\Users\karan\IdeaProjects\Cryptography\src>
```

Discuss Improvements:

Key Management: In my implementation, I used a password-based approach to generate the secret key. However, for enhanced security, a key management system could be implemented.

Error Handling: The program includes basic error handling with informative messages. To improve error handling, more specific error codes and descriptions could be provided.

Security Considerations: While the current implementation showcases the encryption process, it's important to note that the security of the encryption method depends on the strength of the key and algorithm used. Implementing additional security measures such as key strengthening could be considered.

Part 2: Demonstrate Chosen-Plaintext Attack (CPA) Security

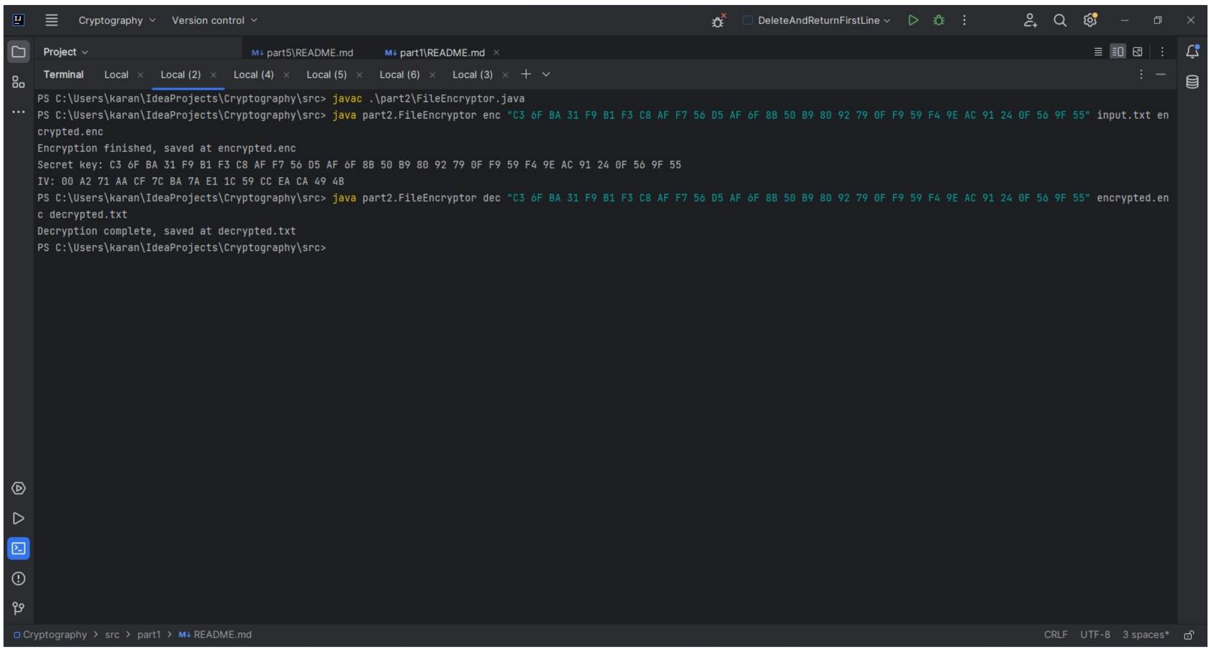
Introduction: Part 2 of the assignment focuses on enhancing the chosen-plaintext attack (CPA) security of the program. In this part, the program's ability to successfully decrypt a previously encrypted file and generate different ciphertexts for each encryption operation is demonstrated.

Implementation Steps:

1. Modify the FileEncryptor program to accept a secret key as a base64 string for encryption and decryption.
2. Update the decryption process to not require specifying the IV. The IV now acts as a salt.
3. Incorporate a method to generate the same ciphertext when encrypting the same plaintext multiple times.
4. Utilize the hexdump utility with the -b flag to showcase different ciphertexts upon multiple encryption runs.

Screenshots:

- Encrypting plaintext file



```
PS C:\Users\karan\IdeaProjects\Cryptography\src> javac .\part2\FileEncryptor.java
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part2.FileEncryptor enc "C3 6F BA 31 F9 B1 F3 C8 AF F7 56 D5 AF 6F 8B 50 B9 80 92 79 0F F9 59 F4 9E AC 91 24 0F 56 9F 55" input.txt en
crypted.enc
Encryption finished, saved at encrypted.enc
Secret key: C3 6F BA 31 F9 B1 F3 C8 AF F7 56 D5 AF 6F 8B 50 B9 80 92 79 0F F9 59 F4 9E AC 91 24 0F 56 9F 55
IV: 00 A2 71 AA CF 7C BA 7A E1 1C 59 CC EA CA 49 4B
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part2.FileEncryptor dec "C3 6F BA 31 F9 B1 F3 C8 AF F7 56 D5 AF 6F 8B 50 B9 80 92 79 0F F9 59 F4 9E AC 91 24 0F 56 9F 55" encrypted.en
c decrypted.txt
Decryption complete, saved at decrypted.txt
PS C:\Users\karan\IdeaProjects\Cryptography\src>
```

Discuss Improvements:

Key Usage and Storage: In the current implementation, the program accepts the secret key as a base64 string. To enhance security, the design could be improved to handle keys securely, such as utilizing a key management system or hardware security module.

IV Management: In this part, the IV serves as a salt and does not need to be kept secret. However, a more advanced implementation could consider encrypting the IV along with the ciphertext for added security.

Output Consistency: While the program generates different ciphertexts for the same plaintext upon multiple encryption runs, ensuring consistent ciphertext generation could be important for certain use cases. Design improvements could focus on maintaining this consistency.

Part 3: Generating a Secret Key from a Password

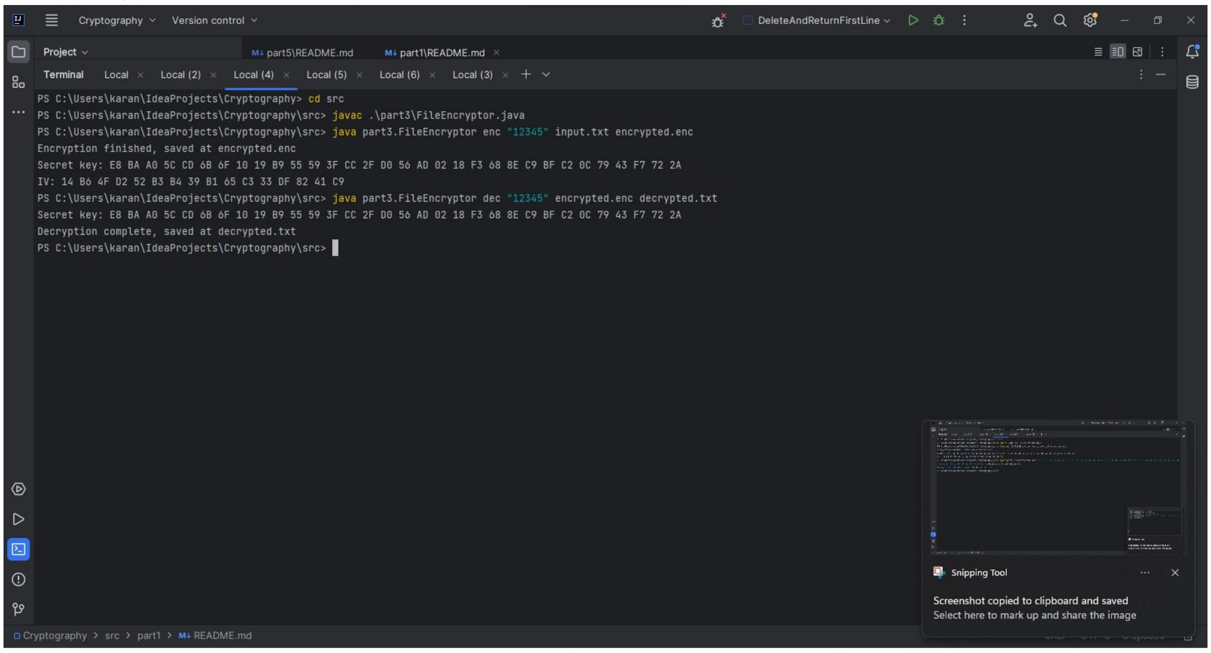
Introduction: Part 3 focuses on enhancing security by generating a secret key from a user-provided password. This approach improves password-based encryption by incorporating salt, iterations, and hashing, in line with NIST and OWASP recommendations.

Implementation Steps:

1. Modify the FileEncryptor program to accept a password as an input for encryption and decryption.
2. Implement password-based key derivation using a recommended algorithm (e.g., PBKDF2).
3. Incorporate the concept of salt and iteration count to strengthen the generated key.
4. Output the secret key derived from the password for marking purposes.

Screenshots:

- Encrypting plaintext file with password



```
PS C:\Users\karan\IdeaProjects\Cryptography> cd src
PS C:\Users\karan\IdeaProjects\Cryptography\src> javac .\part3\FileEncryptor.java
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part3.FileEncryptor enc "12345" input.txt encrypted.enc
Encryption finished, saved at encrypted.enc
Secret key: E8 BA A0 5C CD 6B 6F 10 19 B9 55 59 3F CC 2F D0 56 AD 02 18 F3 68 8E C9 BF C2 0C 79 43 F7 72 2A
IV: 14 B6 4F D2 52 B3 B4 39 B1 65 C3 33 DF 82 41 C9
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part3.FileEncryptor dec "12345" encrypted.enc decrypted.txt
Secret key: E8 BA A0 5C CD 6B 6F 10 19 B9 55 59 3F CC 2F D0 56 AD 02 18 F3 68 8E C9 BF C2 0C 79 43 F7 72 2A
Decryption complete, saved at decrypted.txt
PS C:\Users\karan\IdeaProjects\Cryptography\src>
```

Discuss Improvements:

Password Complexity: While password-based key derivation significantly enhances security, the strength of the generated key depends on the password's complexity. Educating users about creating strong passwords can further bolster the encryption scheme's security.

Algorithm Choices: While PBKDF2 is a recommended algorithm for password-based key derivation, staying updated with the latest advancements and considering more modern algorithms like Argon2 could improve security.

Iterations and Performance: The number of iterations impacts the key's strength and the encryption/decryption performance. Balancing security with performance is crucial. Dynamic iteration adjustment based on hardware capabilities could be explored.

Salting Strategy: Considering an effective salting strategy, such as generating a random salt for each encryption and securely storing the salt along with the ciphertext, could enhance security against rainbow table attacks.

Part 4: Designing for Changes in Key Length and Algorithms

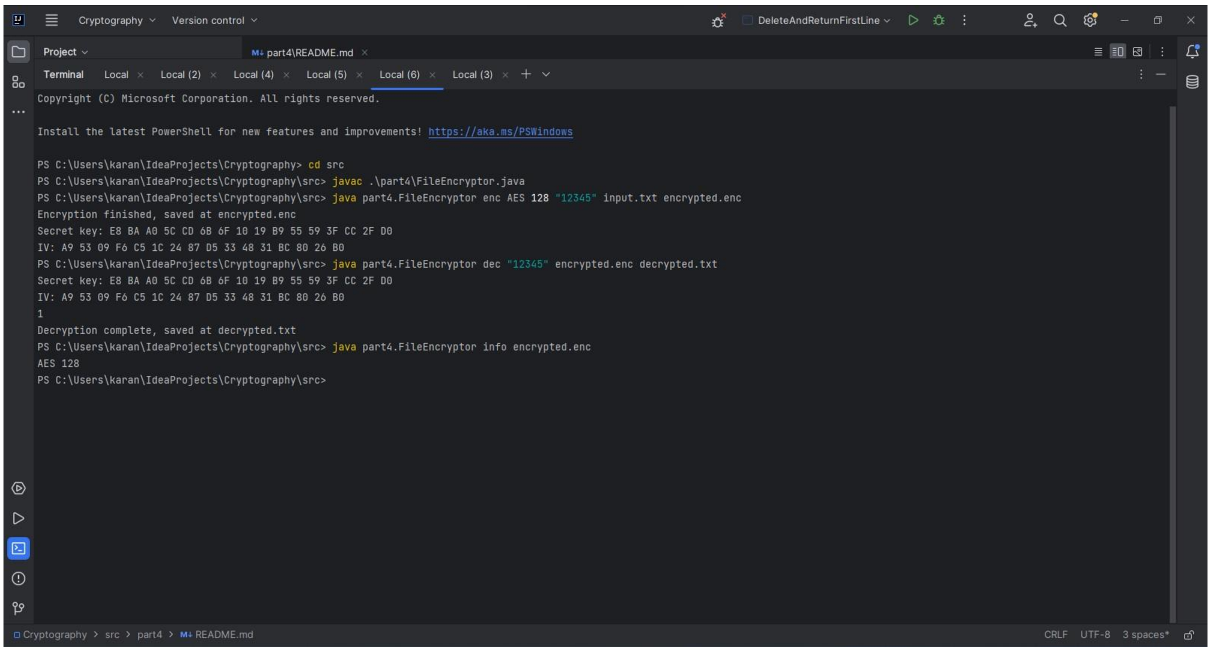
Introduction: Part 4 focuses on adapting to changes in encryption algorithms and key lengths while ensuring compatibility with legacy files. The implementation allows users to specify encryption algorithms and key lengths, embeds metadata in encrypted files, and uses this metadata during decryption.

Implementation Steps:

1. Modify the FileEncryptor program to accept user-specified algorithm and key length.
2. Implement a metadata embedding mechanism to record the algorithm and key length used for encryption.
3. During encryption, include metadata in the encrypted file.
4. During decryption, read metadata from the file to determine the decryption algorithm and key length.
5. Allow users to query the metadata from encrypted files.

Screenshots:

- Encrypting plaintext file with user-specified algorithm and key length



```
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\karan\IdeaProjects\Cryptography> cd src
PS C:\Users\karan\IdeaProjects\Cryptography\src> javac .\part4\FileEncryptor.java
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part4.FileEncryptor enc AES 128 "12345" input.txt encrypted.enc
Encryption finished, saved at encrypted.enc
Secret key: E8 BA A0 5C CD 6B 6F 10 19 B9 55 59 3F CC 2F D0
IV: A9 53 09 F6 C5 1C 24 87 D5 33 48 31 BC 80 26 B0
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part4.FileEncryptor dec "12345" encrypted.enc decrypted.txt
Secret key: E8 BA A0 5C CD 6B 6F 10 19 B9 55 59 3F CC 2F D0
IV: A9 53 09 F6 C5 1C 24 87 D5 33 48 31 BC 80 26 B0
1
Decryption complete, saved at decrypted.txt
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part4.FileEncryptor info encrypted.enc
AES 128
PS C:\Users\karan\IdeaProjects\Cryptography\src>
```

Discuss Improvements:

Metadata Integrity: Ensuring the integrity of embedded metadata is crucial. Implementing checksums or digital signatures for metadata can prevent tampering and provide assurance about the authenticity of the metadata.

Algorithm Agility: As encryption algorithms evolve, incorporating an algorithm agility mechanism that allows the software to adapt to new algorithms while maintaining compatibility with older ones is important.

Key Length Recommendations: Prompting users with recommended key lengths based on the selected algorithm can enhance security by guiding them towards optimal choices.

Metadata Storage: While embedding metadata is suitable for small files, handling metadata for larger files could be challenging. Consider alternative strategies such as storing metadata separately in a secure location.

Part 5: Vulnerable Random Values Generator

Introduction: Part 5 involves designing and implementing a vulnerable Pseudorandom Number Generator (PNG) class and demonstrating its impact on the cryptosystem. The goal is to showcase how using a weak random number generator can compromise the security of encryption by generating predictable IVs.

Design of vPNG Class:

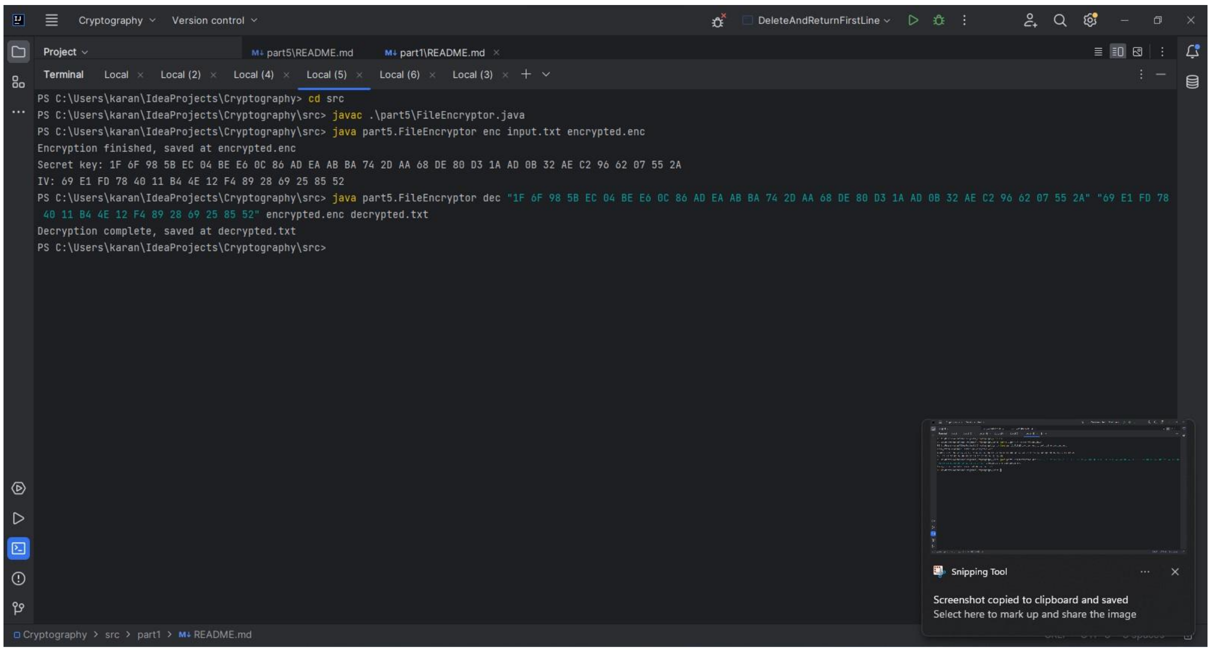
1. Create a class named **vPNG** with a constructor that takes a seed value as an input.
2. Use the seed to initialize the random number generator.
3. Implement a **next()** function that returns the next byte value in the sequence generated by the random number generator.

Demonstration Steps:

1. Implement a use-case where the **vPNG** class generates IV values for encryption in Part 1 of the assignment.
2. Encrypt a file using the vulnerable IVs.
3. Show how the use of weak IVs can lead to predictability in ciphertexts.

Screenshots:

- sGenerating IVs using vPNG for encryption



```
PS C:\Users\karan\IdeaProjects\Cryptography> cd src
PS C:\Users\karan\IdeaProjects\Cryptography\src> javac .\part5\FileEncryptor.java
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part5.FileEncryptor enc input.txt encrypted.enc
Encryption finished, saved at encrypted.enc
Secret key: 1F 6F 98 5B EC 04 BE E6 0C 86 AD EA AB BA 74 2D AA 68 DE 80 D3 1A AD 0B 32 AE C2 96 62 07 55 2A
IV: 69 E1 FD 78 40 11 B4 4E 12 F4 89 28 69 25 85 52
PS C:\Users\karan\IdeaProjects\Cryptography\src> java part5.FileEncryptor dec "1F 6F 98 5B EC 04 BE E6 0C 86 AD EA AB BA 74 2D AA 68 DE 80 D3 1A AD 0B 32 AE C2 96 62 07 55 2A" "69 E1 FD 78
 40 11 B4 4E 12 F4 89 28 69 25 85 52" encrypted.enc decrypted.txt
Decryption complete, saved at decrypted.txt
PS C:\Users\karan\IdeaProjects\Cryptography\src>
```

Impact Demonstration: The use of a vulnerable PNG for generating IVs weakens the security of the encryption. Attackers can predict IV values, leading to potential vulnerabilities such as chosen-plaintext attacks.

Discuss Improvements:

Cryptographically Secure RNG: Use cryptographically secure random number generators provided by libraries. They are designed to produce unpredictable and unbiased values suitable for cryptographic applications.

Entropy Enhancement: Incorporate additional sources of entropy, such as system events, mouse movements, and keyboard inputs, to enhance the unpredictability of random values.

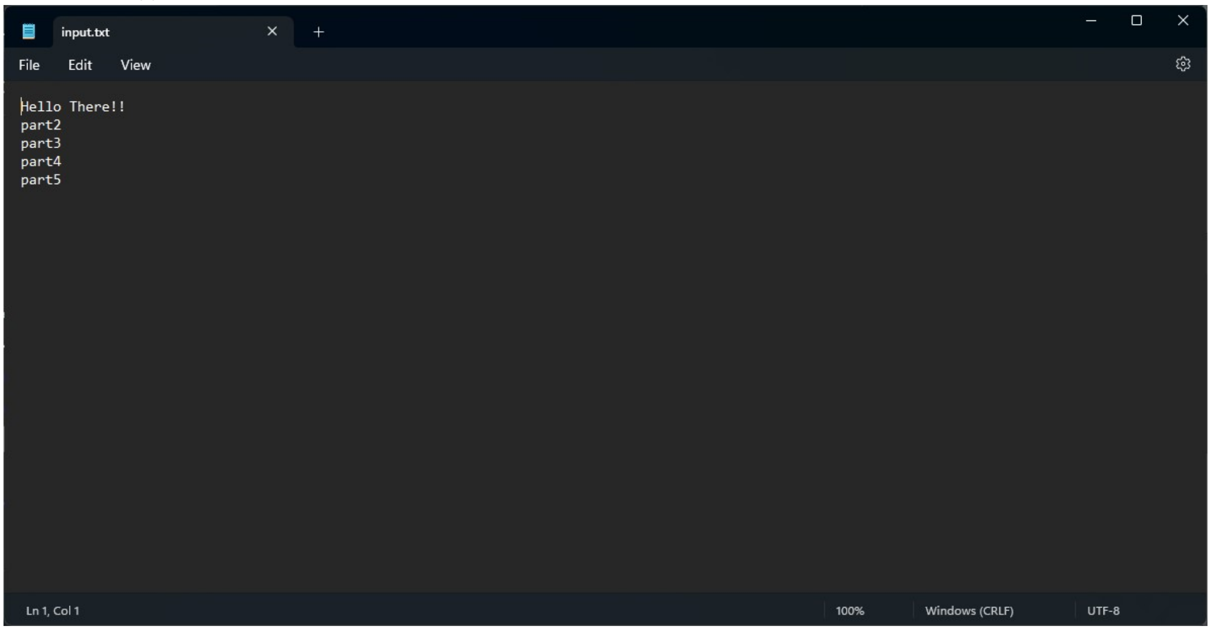
Seed Management: Implement secure methods for seed generation, such as using true random sources or combining multiple sources of randomness.

Regular Updates: Periodically update the random seed to prevent patterns from emerging in the generated values.

By implementing the above improvements, the vulnerability of the random number generator can be significantly reduced.

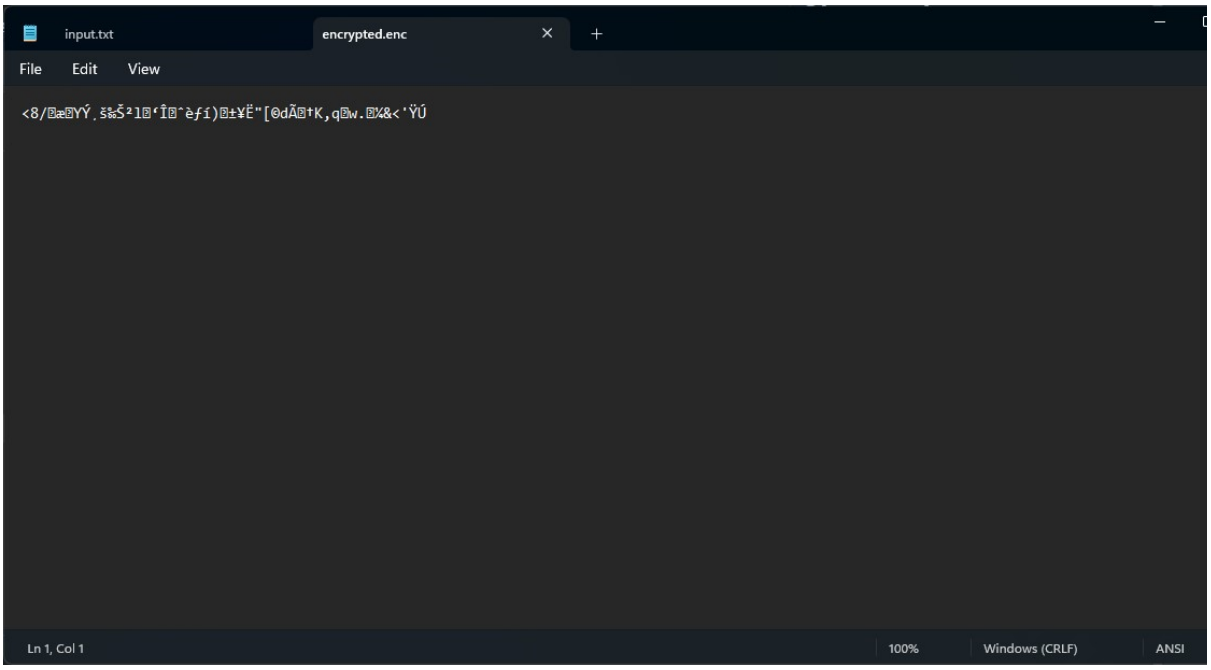
Encrypted and Decrypted File

- File to encrypt:



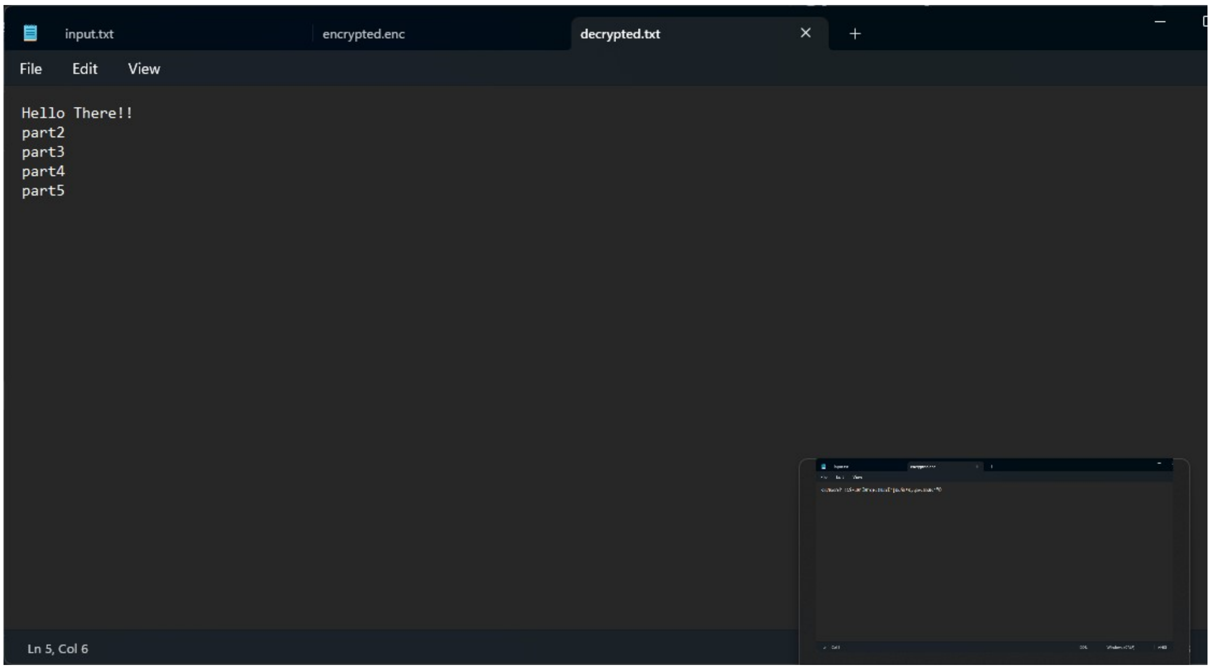
```
Hello There!!
part2
part3
part4
part5
```

- Encrypted file:



```
<8/□æ□YÝ¸š‰Š²1□‘Î□ˆèƒí)□±¥Ë"[ϴdÃ□†K,q□w.□¼&<'ŸÚ
```

- Decrypted File:



```
Hello There!!
part2
part3
part4
part5
```