**CMPG 215 Encryption Project Documentation**

**Composition of general algorithm**

The algorithm uses the Fernet library to encrypt and decrypt the different types of files and the key is also generated using this library. The password entered servers as a verification key. After the content of the file is encrypted, the encrypted contents are stored back into the same file. Therefore, no duplicate files of the encrypted files are created.

**Advantages**

* Security: Fernet is a strong encryption algorithm designed to provide high security to data.
* Compatibility: Fernet library is available in many programming languages and is supported by different computer systems.
* Simplicity: Fernet is easy to use and integrate into Python applications, with a simple API for encrypting and decrypting data.
* Key Management**:** Fernet uses symmetric encryption keys that can be easily stored and managed by the application.

**Disadvantages**

* Key Exchange**:** Like all shared encryption algorithms, Fernet requires both the sender and receiver to have access to the same key. Sometimes changing security keys can be difficult.
* Restricted use**:** Fernet may not be suitable for all uses**.** For example, it may not be suitable for situations where data integration is required or where keys need to be changed frequently.
* Performance: While Fernet is generally fast and efficient, it may not be the best choice for high-performance applications that require short-term encryption and decryption.
* No perfect forward security**:** Like other shared encryption algorithms, Fernet does not offer perfect forward secrecy. This means that once a key is compromised, all data encrypted with that key will also be compromised.

**Detail description of own algorithm**

The own algorithm uses a custom encryption and decryption function based on the XOR bitwise operation. It does not use any standardized encryption or decryption algorithm like AES or RSA. The key is used as a XOR mask to encrypt and decrypt the data byte-by-byte. To decrypt data that has been encrypted using this algorithm, the same key that was used to encrypt the data must be used as a XOR mask to reverse the process.

The encryption and decryption methods explained in detail

The **‘OwnEncryption**()’ function takes two parameters: ‘**file\_path’**, which is the path to the file that needs to be encrypted, and key, which is the encryption key. The if statement compares the hashed version of the user-entered password (**myEntry.get**()) with the hashed version of the stored password (**Password**). The **hash\_password** function is used to hash both passwords for comparison. If the hashed passwords match, the code inside the if block is executed. Otherwise, an error message is displayed using messagebox.showinfo.

Inside the if block, the file specified by file\_path is opened in binary mode ("rb") using a with statement. The file contents are read and stored in the data variable as a bytearray

The key parameter is checked to ensure it is of type string. If it is not, it is encoded as UTF-8.

The for loop iterates over each index-value pair in the data bytearray using the enumerate function. For each pair, the index is used to retrieve the corresponding byte from the key using the modulo operator (%) to ensure the key is repeated if it is shorter than the data. The byte from the key is then XORed with the byte from the data, resulting in an encrypted byte.

A new file path is created by appending the extension .enc to the original file\_path. This will be the path of the encrypted file.

The encrypted data is written to the new file path using a with statement and the file opened in binary mode ("wb").

The original file specified by file\_path is removed using os.remove to delete it after encryption.

If the passwords do not match (the else block), a message box is displayed indicating that the password entered is incorrect, and the encryption process is halted.

The `OwnEncryption` function is responsible for encrypting a file. It achieves this by reading the file's contents, encrypting each byte individually using the XOR operation with the provided encryption key. The encrypted data is then written to a new file with the extension `.enc`. After encryption, the original file is deleted. Before the encryption process begins, the function verifies the password by comparing the hashed versions of the entered password and the stored password.

The ‘**OwnDecryption**()’ function is very similar to ‘**OwnEncryption**()’, but it reverses the XOR operation to decrypt the data. It takes two parameters: **file\_path** (the path of the file to be decrypted) and key (the decryption key). The first if statement compares the hashed version of the user-entered password (**myEntry.ge**t()) with the hashed version of the stored password (Password). This ensures that the correct password is provided before proceeding with decryption. The **hash\_password** function is used to hash both passwords for comparison.

If the hashed passwords match, the code inside the if block is executed. Otherwise, an error message is displayed using messagebox.showinfo, indicating that the provided password is incorrect.

Inside the if block, the file specified by **file\_path** is opened in binary mode ("rb") using a with statement. The file contents are read and stored in the data variable as a bytearray.

The key parameter is checked to ensure it is of type string. If it is not, it is encoded as UTF-8.

The for loop iterates over each index-value pair in the data bytearray using the enumerate function. For each pair, the index is used to retrieve the corresponding byte from the key using the modulo operator (%) to ensure the key is repeated if it is shorter than the data. The byte from the key is then XORed with the byte from the data, resulting in a decrypted byte.

The decrypted data is written back to the original file path specified by file\_path using a with statement and the file opened in binary mode ("wb"). This overwrites the encrypted contents with the decrypted data.

The original file extension is retrieved using os.path.splitext, which splits the file path into the base name and the extension. The decrypted file path is obtained by removing the .enc extension from the base name.

The original encrypted file specified by file\_path is renamed to the decrypted file path using os.rename. This effectively replaces the encrypted file with the decrypted file, as the file path is modified.

If the passwords do not match (the else block), a message box is displayed indicating that the provided password is incorrect. The decryption process is halted in this case.

In essence, the `OwnDecryption` function first checks the password's validity to ensure the correct password is provided before starting the decryption process. It then proceeds to read the encrypted file, decrypting each byte individually using the XOR operation and the decryption key. The decrypted data replaces the contents of the original file. Additionally, the file extension is adjusted to signify that it has undergone decryption.

While XOR can provide some level of security, it is generally considered to be a weak encryption method that should not be relied on for strong security. This is because XOR is vulnerable to several types of attacks, including brute force attacks, statistical attacks, and known plaintext attacks. Furthermore, the implementation of hashing function provides enhanced security by converting the password into a fixed-size hashed value. Hash functions are deliberately created to be computationally demanding to reverse, rendering the task of retrieving the original password from its hashed form exceedingly arduous. Moreover, the inclusion of a salt value adds an extra layer of uniqueness to each password's hash. This uniqueness prevents attackers from utilizing precomputed hash tables (rainbow tables) to expedite password cracking attempts. As a result, this approach strengthens the overall security of password storage and verification processes.

**Comparison of own algorithm with other algorithms**

The own algorithm is a custom encryption algorithm based on the XOR bitwise operation, has several differences and limitations when compared to other standardized encryption algorithms such as AES and RSA.

* Security: While XOR can provide some security, it is generally considered a weak encryption method and should not be relied upon for security. AES and RSA, on the other hand, are considered strong encryption methods and are widely used in businesses that require high security.
* Key length: Special algorithms use a key for encryption and decryption whose length is limited by the size of the encrypted data. However, AES and RSA use longer keys (128, 192, or 256 bits for AES**,** usually 1024 or 2048 bits for RSA), which makes encryption harder to crack**.**
* Standardization: AES and RSA are widely used and accepted encryption standards, while the own algorithms are proprietary implementations that have not undergone the same level of analysis and testing.
* Performance: Since it is a simple algorithm that requires less computational power to encrypt and decrypt the data sheet, the own algorithm will outperform AES and RSA in some cases**.**

Overall, the own algorithm may be fine for simple applications where security is not a major concern but is not recommended for critical situations**.** In this case, standard encryption algorithms such as AES and RSA should be used**.**  
  
**References**

The following links are served as resources as they helped in understanding and completing the program.

<https://towardsdatascience.com/encrypt-and-decrypt-files-using-python-python-programming-pyshark-a67774bbf9f4>

<https://youtu.be/i-h0CtKde6w>

<https://youtu.be/TGqCy1_kDv>

<https://youtu.be/ibf5cx221hk>

<https://pythonistaplanet.com/fernet/>

<https://cryptography.io/en/3.4.2/>

<https://pypi.org/project/cryptography/>