AMRITA VISHWA VIDYAPEETHAM

SCHOOL OF ARTIFICIAL INTELLIGENCE

PROFFESSIONAL ELECTIVE 1:

APPLIED CRYPTOGRAPHY

MID-TERM PROJECT

B.TECH

CSE-AI(Semester-5)

User Based File & Image Encryption/Decryption Using AES

Under the supervision of:

Dr.K.P Soman AND Dr.Sunil Kumar S

SUBMITTED BY

GROUP-2

JAIDEV K (CB.EN.U4AIE21117)

J SAI CHANDANA(CB.EN.U4AIE21118)

PRANISH KUMAR .M(CB.EN.U4AIE21137)

T.CHARISHMA (CB.EN.U4AIE21169)

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our esteemed faculty for their invaluable guidance and support throughout our cryptography project on User Authenticated Image & File Encryption/Decryption. Your expertise and mentor-ship were instrumental in shaping our project's success. Your dedication to our learning and development has been a source of inspiration. Thank you for your unwavering commitment to our academic growth. We deeply appreciate your contributions to our team's journey.

TABLE OF CONTENTS:

ABSTRACT:

This research introduces a comprehensive solution for information security by developing a secure file and image encryption/decryption system utilizing the Advanced Encryption Standard (AES) algorithm. The system incorporates user authentication through standard methods like username and password validation, providing an additional layer of security. Upon successful authentication, users can access encryption and decryption functionalities, with the AES algorithm ensuring the transformation of plaintext files and images into unreadable ciphertext.

To enhance user experience and security, the system integrates key management techniques, allowing users to customize encryption key preferences. Additionally, the system incorporates integrity checks and error handling mechanisms to ensure the reliability of encrypted data, detecting and rectifying potential corruption or tampering.

The user interface is designed for intuitiveness, enabling users to navigate the system easily, upload files or images, and initiate cryptographic operations effortlessly. In conclusion, this research offers a robust and user-friendly solution that combines user authentication and AES encryption to fortify data confidentiality while addressing key management, integrity, and user experience considerations in the realm of information security.

AES (Advanced Encryption Standard)

Here's a brief overview of AES (Advanced Encryption Standard) and the steps involved in both encryption and decryption:

Overview:

Type: Symmetric Key Encryption Algorithm

Key Sizes: Supports key sizes of 128, 192, and 256 bits.

Block Size: Operates on fixed-size blocks of 128 bits.

Rounds: The number of rounds varies based on the key size (10 rounds for 128-bit key, 12 rounds for 192-bit key, and 14 rounds for 256-bit key).

Algorithm Structure: Utilizes a substitution-permutation network (SPN) structure.

Steps for AES Encryption:

Start Encryption:

Begin the encryption process.

User Authentication:

Verify the user's identity through standard authentication methods.

Input: Plaintext:

Obtain the original data (plaintext) that needs to be encrypted.

Key Expansion:

Expand the user-provided key into a set of round keys for each encryption round.

Initial Round:

Perform the initial encryption round:

SubBytes: Substitute each byte with a corresponding value from the S-box.

ShiftRows: Shift rows of the state matrix.

MixColumns: Mix the columns of the state matrix.

AddRoundKey: XOR the state matrix with the round key.

Main Rounds (Repeated):

Perform a series of rounds (9 rounds for 128-bit key, 11 rounds for 192-bit key, and 13 rounds for 256-bit key):

SubBytes: Byte substitution using the S-box.

ShiftRows: Row-wise shifting of the state matrix.

MixColumns: Column mixing of the state matrix.

AddRoundKey: XOR the state matrix with the round key.

Final Round:

Conclude the encryption process with a final round (similar to main rounds but without MixColumns):

SubBytes: Byte substitution.

ShiftRows: Row-wise shifting.

AddRoundKey: XOR the state matrix with the final round key.

Ciphertext Output:

The result after the final round is the encrypted data (ciphertext).

End Encryption:

Complete the encryption process.

Steps for AES Decryption:

Start Decryption:

Begin the decryption process.

User Authentication:

Verify the user's identity through standard authentication methods.

Input: Ciphertext:

Obtain the encrypted data (ciphertext) that needs to be decrypted.

Key Expansion:

Expand the user-provided key into a set of round keys for each decryption roud.

Inverse Final Round:

Perform the inverse of the final encryption round:

AddRoundKey: XOR the ciphertext with the final round key.

InverseShiftRows: Reverse row-wise shifting.

InverseSubBytes: Reverse byte substitution.

Inverse Main Rounds (Repeated):

Perform a series of rounds (in reverse order compared to encryption):

AddRoundKey: XOR the state matrix with the round key.

InverseMixColumns: Reverse column mixing.

InverseShiftRows: Reverse row-wise shifting.

InverseSubBytes: Reverse byte substitution.

Inverse Initial Round:

Perform the inverse of the initial encryption round:

AddRoundKey: XOR the state matrix with the round key.

InverseMixColumns: Reverse column mixing.

InverseShiftRows: Reverse row-wise shifting.

InverseSubBytes: Reverse byte substitution.

Plaintext Output:

The result after the inverse initial round is the decrypted data (plaintext).

End Decryption

Image & File Encryption/Decryption

Encryption :

a secure file encryption process that involves user input, leveraging industry-standard practices to ensure data confidentiality.

- The process begins with user initiation, where a file and a secure encryption key or password are provided. The file is then uploaded to the server using the Multer middleware, ensuring a secure and smooth transfer.
- Key generation follows, employing the scrypt function to derive a cryptographic key from the user-provided password, thereby enhancing the security of the encryption process.
- To further fortify security, a random Initialization Vector (IV) is generated, crucial for the encryption algorithm's integrity. The robust AES-256-CFB encryption algorithm is then applied to the file content using the generated key and IV.
- The IV is concatenated with the encrypted content to create a final encrypted data package.

This encrypted data is stored securely, preventing unauthorized access.

- The file is named appropriately and saved in a designated directory named "encrypted_files." A response is promptly sent to the user, confirming the successful encryption and providing a secure download link for retrieving the encrypted file.
- Logging mechanisms are incorporated to record essential details of the encryption event, facilitating auditing and debugging.

Password Hashing & Salt function Using Bcrypt

Hashing:

When a user creates or updates their password, the bcrypt function hashes the password. Hashing is a one-way process that transforms the password into a fixed-length string of characters.

Salting:

bcrypt uses a technique called salting to enhance security. A unique salt is generated for each password. This salt is then combined with the user's password before hashing. Salting is crucial because it prevents attackers from using precomputed tables (rainbow tables) to quickly look up the hash value for a given password.

Work Factor (Cost Parameter):

bcrypt allows the specification of a cost parameter (also known as the work factor). This parameter determines the computational cost of the hashing algorithm. A higher cost makes it more computationally intensive, thereby making brute-force and rainbow table attacks more difficult.

Storage:

The hashed password, along with the salt and cost parameter, is then stored in the database. The typical format is a string that includes the hash, salt, and cost factor. For example, a bcrypt hash might look like this:

$2a$10$SALTSTRINGHASHSTRING

- "$2a$" indicates the use of the bcrypt algorithm.
- "10" is the cost factor.
- "SALTSTRING" is the salt.
- "HASHSTRING" is the hashed password

Size of Salt :

- the size of the salt in a bcrypt hash does not vary based on the size of the hashed password. In bcrypt, the salt is a fixed-size component, and it doesn't depend on the length of the password being hashed.
- Typically, the salt size in bcrypt is 128 bits (16 bytes). Regardless of whether the password is short or long, the salt will remain the same size.
- The fixed-size salt is an important aspect of bcrypt's design, contributing to its security by ensuring a consistent level of randomness and complexity is added to each password hash.
- The salt (SALTSTRING) is always 22 characters in length, regardless of the length of the original password being hashed.

Decryption :

- In the secure file decryption process, user interaction begins with the initiation of decryption by uploading the encrypted file and providing the decryption key or password.
- The encrypted file is securely transmitted to the server, where a cryptographic key is generated using the scrypt function from the user-provided decryption password. This key generation step enhances the overall security of the decryption process. Subsequently, the Initialization Vector (IV), a crucial component for decryption, is extracted from the beginning of the encrypted file.
- The AES-256-CFB decryption algorithm is then applied to the file content, utilizing the generated key and extracted IV. The decrypted content is extracted from this process, saved as a new file in the designated decrypted_files directory, and a response is promptly sent to the user, confirming the successful decryption and providing a secure link to download the decrypted file.

To reinforce security measures, emphasis is placed on the secure generation and handling of cryptographic keys and IVs throughout the decryption process.

- This includes the meticulous generation of the decryption key using the scrypt function, the secure transmission and storage of the encrypted file, and the careful extraction and utilization of the IV during decryption.
- Comprehensive logging mechanisms are implemented, generating log entries that capture relevant details of the decryption event. This serves not only for auditing purposes but also for facilitating debugging activities.
- By prioritizing secure key handling, encrypted file transmission, and robust logging, the decryption process ensures a reliable and secure experience for users seeking to retrieve their decrypted files.

User Authentication

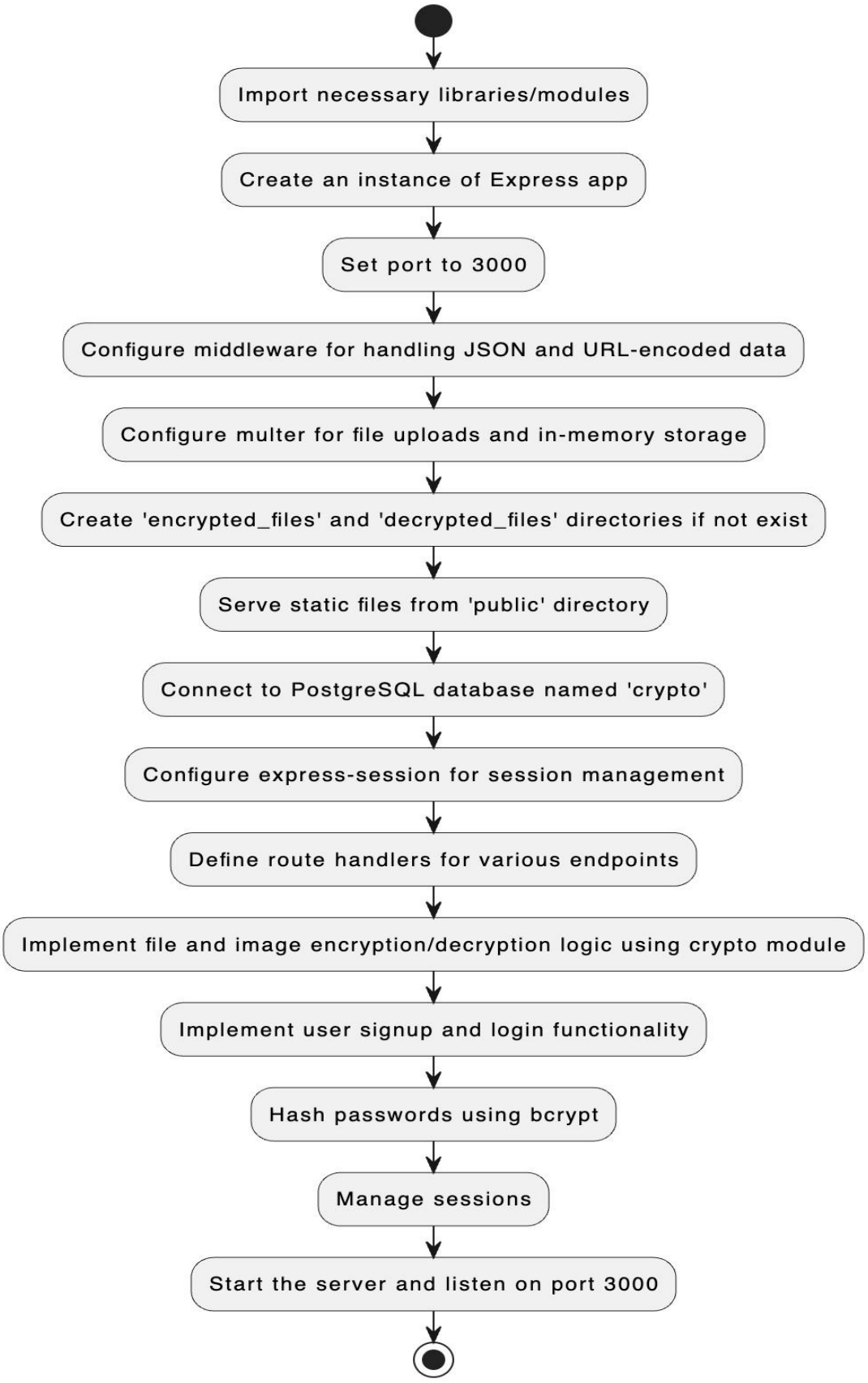
In the user authentication process, users initiate access by providing their credentials, namely, a username and password.

- To validate these credentials, a database query is executed against the PostgreSQL database to retrieve the stored hashed password associated with the provided username.
- The security of this process is fortified through the use of bcrypt, a robust hashing algorithm. Bcrypt is employed to compare the hashed version of the entered password with the stored hashed password in the database.
- If the comparison is successful, signifying a match, the user is successfully authenticated.
- Following authentication, a session is established, often utilizing Express sessions, to persist the user's authentication status. Authenticated users gain access to specific functionalities and secure areas of the application, ensuring that only authorized individuals can interact with sensitive features.

The implementation of bcrypt for secure password hashing stands out as a crucial security measure in this authentication process.

- This approach enhances the protection of user credentials and prevents unauthorized access, even in the event of a potential compromise of the database.
- The emphasis on bcrypt underscores the commitment to safeguarding user data by employing industry-standard security practices, ensuring a resilient and secure authentication mechanism for users interacting with the application.

BLOCK DIAGRAM:



CODE :

```
const { Client } = require('pg');
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const bcrypt = require('bcrypt');
const crypto = require('crypto');
const session = require('express-session');
const app = express();
const port = 3000;

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

const encryptedFolderPath = 'encrypted_files';
const decryptedFolderPath = 'decrypted_files';

if (!fs.existsSync(encryptedFolderPath)) {
  fs.mkdirSync(encryptedFolderPath);
}

if (!fs.existsSync(decryptedFolderPath)) {
  fs.mkdirSync(decryptedFolderPath);
}
```

```
app.use(express.static(path.join(__dirname, 'public')));

const db = new Client({
  user: "postgres",
  host: "localhost",
  database: "crypto",
  password: "jsc2003",
  port: 5432,
});
db.connect();

app.use(session({
  secret: 'jsc',
  resave: false,
  saveUninitialized: false,
  // ...other options as needed
}));
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.get('/login', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'login.html'));
});

app.get('/signup', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'signup.html'));
});

app.get('/dashboard', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'dashboard.html'));
});
```

```
app.get('/file-encryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'file-encryption.html'));
});

app.get('/file-decryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'file-decryption.html'));
});

app.get('/download-decryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'download-decryption.html'));
});

app.get('/image-en-dec', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'image-en-dec.html'));
});

app.get('/image-encryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'image-encryption.html'));
});

app.get('/download-image-encryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'download-image-encryption.html'));
});

app.get('/image-decryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'image-decryption.html'));
});

app.get('/download-image-decryption', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'download-image-decryption.html'));
});
```

```
95  app.get('/download/:filename', (req, res) => {
96    const filename = req.params.filename;
97    const filePath = path.join(__dirname, 'decrypted_files', filename);
98
99    if (fs.existsSync(filePath)) {
100     res.download(filePath, (err) => {
101       if (err) {
102         console.error('Error downloading file:', err);
103         res.status(500).send('Error downloading file.');
104       }
105     });
106   } else {
107     res.status(404).send('File not found.');
108   }
109 });
110
111 app.get('/download-encrypted/:filename', (req, res) => {
112   const filename = req.params.filename;
113   const filePath = path.join(__dirname, 'encrypted_files', filename);
114
115   if (fs.existsSync(filePath)) {
116     res.download(filePath, (err) => {
117       if (err) {
118         console.error('Error downloading file:', err);
119         res.status(500).send('Error downloading file.');
120       }
121     });
122   } else {
123     res.status(404).send('File not found.');
124   }
125 });
126
127 app.post('/encrypt-file', upload.single('fileToUpload'), (req, res) => {
```

```
const filePassword = req.body.filePassword;
const fileContent = req.file.buffer;

const algorithm = 'aes-256-cfb';
const key = crypto.scryptSync(filePassword, 'salt', 32);
const iv = crypto.randomBytes(16);
const cipher = crypto.createCipheriv(algorithm, key, iv);

const encryptedContent = Buffer.concat([iv, cipher.update(fileContent), cipher.final()]);

const encryptedFilePath = path.join(encryptedFolderPath, 'encrypted_file.txt');

fs.writeFileSync(encryptedFilePath, encryptedContent);

res.send({
  success: true,
  message: 'File encryption successful!',
  filePath: /download/${path.basename(encryptedFilePath)},
});
);

pp.post('/decrypt-file', upload.single('fileToDecrypt'), (req, res) => {
const filePassword = req.body.filePassword;
const fileContent = req.file.buffer;

const algorithm = 'aes-256-cfb';
const key = crypto.scryptSync(filePassword, 'salt', 32);
const iv = fileContent.slice(0, 16);
const decipher = crypto.createDecipheriv(algorithm, key, iv);

const decryptedContent = Buffer.concat([decipher.update(fileContent.slice(16)), decipher.final()]);

const decryptedFilePath = path.join(decryptedFolderPath, 'decrypted_file.txt');
```

```
161     fs.writeFileSync(decryptedFilePath, decryptedContent);
162
163     res.send({
164       success: true,
165       message: 'File decryption successful!',
166       filePath: /download-decryption/${path.basename(decryptedFilePath)},
167     });
168   });
169
170   app.post('/encrypt-image', upload.single('imageToUpload'), (req, res) => {
171     const imagePassword = req.body.imagePassword;
172     const imageData = req.file.buffer;
173
174     const algorithm = 'aes-256-cfb';
175     const key = crypto.scryptSync(imagePassword, 'salt', 32);
176     const iv = crypto.randomBytes(16);
177     const cipher = crypto.createCipheriv(algorithm, key, iv);
178
179     const encryptedImageData = Buffer.concat([iv, cipher.update(imageData), cipher.final()]);
180
181     const encryptedImagePath = path.join(encryptedFolderPath, 'encrypted_image.jpg');
182
183     fs.writeFileSync(encryptedImagePath, encryptedImageData);
184
185     res.send({
186       success: true,
187       message: 'Image encryption successful!',
188       filePath: /download-encrypted/${path.basename(encryptedImagePath)},
189     });
190   });
191
192   // Add this route for image decryption
193   app.get('/download-decrypted-image/:filename', (req, res) => {
```

```
const filename = req.params.filename;
const filePath = path.join(__dirname, 'decrypted_files', filename);

console.log('Download Decrypted Image Path:', filePath);

if (fs.existsSync(filePath)) {
  res.download(filePath, (err) => {
    if (err) {
      console.error('Error downloading image:', err);
      res.status(500).send('Error downloading image.');
    }
  });
} else {
  res.status(404).send('Image not found.');
}
);

pp.post('/decrypt-image', upload.single('imageToDecrypt'), (req, res) => {
const imagePassword = req.body.imagePassword;
const imageData = req.file.buffer;

const algorithm = 'aes-256-cfb';
const key = crypto.scryptSync(imagePassword, 'salt', 32);
const iv = imageData.slice(0, 16);
const decipher = crypto.createDecipheriv(algorithm, key, iv);

const decryptedImageData = Buffer.concat([decipher.update(imageData.slice(16)), decipher.final()]);

const decryptedImagePath = path.join(decryptedFolderPath, 'decrypted_image.jpg');
fs.writeFileSync(decryptedImagePath, decryptedImageData);

res.send({
  success: true,
```

```
    message: 'Image decryption successful',
    filePath: /download-image-decryption/${path.basename(decryptedImagePath)},
  });
});

app.post('/signup', (req, res) => {
  const { username, password } = req.body;

  // Encrypt the password using a secure hashing algorithm
  const hashedPassword = bcrypt.hashSync(password, 10);

  // Insert into the new table (no need to provide ID explicitly)
  db.query(
    INSERT INTO new_users (username, password) VALUES ($1, $2),
    [username, hashedPassword],
    (err, result) => {
      if (err) {
        if (err.code === '23505') {
          res.status(400).send('Username already exists'); // Handle duplicate username error
        } else {
          console.error(err);
          res.status(500).send('Error saving user'); // Handle other database errors
        }
      } else {
        res.redirect('/dashboard.html'); // Redirect on successful signup
      }
    }
  );
});


app.post('/login', (req, res) => {
  const { username, password } = req.body;
```

```
db.query(
  'SELECT * FROM new_users WHERE username = $1',
  [username],
  (err, result) => {
    if (err) {
      console.error(err);
      res.status(500).send('Error checking credentials');
    } else if (result.rows.length === 0) {
      res.status(401).send('Invalid username or password');
    } else {
      const user = result.rows[0];
      bcrypt.compare(password, user.password, (err, isMatch) => {
        if (err) {
          console.error(err);
          res.status(500).send('Error comparing password');
        } else if (isMatch) {
          // Successful login: Set a session cookie or store user info in a session store
          req.session.userId = user.id; // Example using a session store
          res.redirect('/dashboard.html');
        } else {
          res.status(401).send('Invalid username or password');
        }
      });
    }
  }
);
);

pp.listen(port, () => {
 console.log(Server is running at http://localhost:${port});
);
```

FUTURE ENHANCEMENTS

Algorithm Flexibility:

Introduce support for different encryption algorithms, providing users with options based on their security requirements.

Improved UI/UX:

Enhance the user interface for a more intuitive and visually appealing experience. Streamline user interactions and navigation.

Additional File Types:

Expand file handling capabilities to support a broader range of file types.

Cater to diverse user needs for encryption and decryption.

Multi-User Collaboration:

Implement features that enable secure collaboration between multiple users, allowing shared access to encrypted content.

Conclusion

In conclusion, this File and Image Encryption/Decryption Web Application combines cutting-edge technologies with robust security measures. From secure password handling and encryption to database integration, the application ensures the confidentiality and integrity of user data. As we look to the future, potential enhancements aim to provide users with greater flexibility, an improved user experience, and expanded functionality. This project stands as a testament to the commitment to both user-friendliness and stringent security standards in the realm of file and image handling.