FileSafe: Secure File Storage and Encryption Software

Development Project Report

CTEC2451D

JEZIL ISMAIL

P2696270

8400 WORDS

Table of Contents

1. Abstract

The project focuses on the development of a file encryption system using Python. The system allows users to encrypt and decrypt files securely, ensuring confidentiality and integrity of sensitive data. Key features include password protection, cryptographic algorithms, and a user-friendly interface. The project follows a systematic development methodology, incorporating verification and validation techniques to ensure the reliability and effectiveness of the system. Through this project, theoretical concepts in cryptography and software development are applied to address a practical need for secure file management.

2. Introduction

In today's digital age, ensuring the security of sensitive data is paramount. With the increasing threat of cyber-attacks and data breaches, there is a growing demand for robust encryption systems to safeguard confidential information. In response to this need, this project aims to develop a file encryption system that provides users with a reliable and user-friendly solution for securing their files.

The primary objective of the project is to design and implement a software system capable of encrypting and decrypting files using strong cryptographic algorithms. The system will incorporate features such as password protection, file management, and encryption key management to ensure the confidentiality and integrity of user data.

The project will follow a structured development methodology, encompassing the stages of requirements analysis, design, implementation, testing, and evaluation. Throughout the development process, emphasis will be placed on adhering to best practices in software engineering and cryptography to ensure the reliability and security of the system.

By the completion of this project, it is expected to provide a functional file encryption system that meets the specified requirements and demonstrates proficiency in applying theoretical concepts to practical software development. Additionally, the project aims to contribute to the body of knowledge in the field of cybersecurity and encryption systems.

3. Literature Review

3.1 Overview of File Encryption Systems

File encryption systems play a critical role in ensuring the confidentiality and integrity of sensitive data stored on digital devices. In essence, file encryption involves the process of converting plaintext data into ciphertext using cryptographic algorithms, making it unreadable to unauthorized users. This section provides an overview of file encryption systems, highlighting their significance in data security and exploring the evolution of encryption technologies.

Importance of File Encryption in Data Security:

In today's interconnected world, the protection of sensitive information has become increasingly vital. Organizations and individuals alike face numerous threats, including cyber-attacks, data breaches, and unauthorized access to confidential data. File encryption serves as a fundamental safeguard against these threats by rendering data indecipherable to unauthorized users. By encrypting files, sensitive information such as financial records, personal documents, and proprietary business data can be protected from prying eyes and malicious actors.

Moreover, compliance regulations and data privacy laws mandate the implementation of encryption measures to safeguard sensitive data. Regulations such as the General Data Protection Regulation (GDPR) and the Health Insurance Portability and Accountability Act (HIPAA) require organizations to implement encryption as part of their data protection strategies. Failure to comply with these regulations can result in severe penalties and reputational damage.

Evolution of File Encryption Systems:

The evolution of file encryption systems can be traced back to ancient civilizations, where techniques such as secret codes and ciphers were used to conceal sensitive information. However, modern file encryption systems have significantly evolved, driven by advancements in technology and cryptography.

One of the earliest forms of file encryption was the use of symmetric encryption algorithms, where a single key is used for both encryption and decryption. Symmetric encryption systems, such as the Data Encryption Standard (DES) and Advanced Encryption Standard (AES), remain widely used due to their efficiency and effectiveness.

In recent years, asymmetric encryption algorithms have gained prominence, offering enhanced security and key management capabilities. Asymmetric encryption, also known as public-key cryptography, utilizes a pair of keys (public and private) to encrypt and decrypt data. Algorithms such as RSA (Rivest-Shamir-Adleman) and Elliptic Curve Cryptography (ECC) are commonly used in asymmetric encryption systems.

Furthermore, the emergence of quantum computing poses new challenges and opportunities for file encryption. Quantum-resistant encryption algorithms are being developed to mitigate the threat posed by quantum computers, which have the potential to break traditional encryption schemes.

3.2 Review of Existing Solutions

The landscape of file encryption solutions is diverse, encompassing a range of commercial and open-source software designed to protect sensitive data. This section provides a review of existing solutions, evaluating their features, strengths, and weaknesses, and analyzing their relevance to the development of the proposed file encryption system.

Analysis of Commercial File Encryption Software:

Commercial file encryption software offers a comprehensive suite of features tailored to meet the security needs of businesses and individuals. Solutions such as VeraCrypt, BitLocker, and Symantec Endpoint Encryption provide robust encryption algorithms, user-friendly interfaces, and advanced security features.

VeraCrypt, an open-source disk encryption software, offers strong encryption capabilities with support for various encryption algorithms, including AES, Serpent, and Twofish. Its ability to create encrypted containers and partitions makes it a popular choice for securing sensitive data on desktops and external drives. However, VeraCrypt's complexity may pose challenges for novice users, requiring a learning curve to fully utilize its features.

BitLocker, integrated into Microsoft Windows operating systems, provides full-disk encryption for enhanced data protection. Its seamless integration with Windows environments and centralized management capabilities make it an attractive choice for organizations seeking to enforce encryption policies across their network. However, BitLocker's reliance on proprietary encryption algorithms may raise concerns about interoperability and vendor lock-in.

Symantec Endpoint Encryption offers enterprise-grade encryption solutions for endpoint devices, including desktops, laptops, and mobile devices. Its centralized management console allows administrators to enforce encryption policies, manage encryption keys, and monitor compliance across the organization. However, the complexity and cost of deployment may be prohibitive for small businesses and individual users.

Examination of Open-Source File Encryption Tools:

Open-source file encryption tools provide cost-effective solutions for individuals and organizations seeking transparent and auditable encryption solutions. Software such as GnuPG (GNU Privacy Guard) and OpenSSL offer robust encryption capabilities, community-driven development, and interoperability with a wide range of platforms.

GnuPG, a free and open-source implementation of the OpenPGP standard, provides strong encryption and digital signature capabilities for securing files and communications. Its command-line interface and support for various encryption algorithms make it a versatile tool for encrypting files, emails, and documents. However, GnuPG's lack of a graphical user interface (GUI) may deter users accustomed to intuitive interfaces.

OpenSSL, a widely-used cryptographic library, offers encryption and decryption functions for securing data in transit and at rest. Its support for industry-standard encryption algorithms, such as AES and RSA,

makes it a popular choice for developers seeking to integrate encryption into their applications. However, OpenSSL's complex API and documentation may present challenges for developers unfamiliar with cryptographic concepts.

Evaluation of Strengths and Weaknesses:

Commercial file encryption software offers comprehensive features and support, but may be cost-prohibitive for some users. Open-source solutions provide cost-effective alternatives with community-driven development and transparency, but may lack user-friendly interfaces and support options. By evaluating the strengths and weaknesses of existing solutions, developers can identify opportunities to enhance the usability, security, and affordability of the proposed file encryption system.

3.3 Theoretical Background on Encryption Algorithms

Encryption algorithms form the foundation of file encryption systems, providing the mathematical techniques necessary to transform plaintext data into ciphertext. This section explores the theoretical principles behind encryption algorithms, including symmetric and asymmetric encryption, and examines commonly used algorithms such as AES (Advanced Encryption Standard) and RSA (Rivest-Shamir-Adleman).

Overview of Symmetric and Asymmetric Encryption:

Symmetric encryption, also known as secret-key encryption, utilizes a single key for both encryption and decryption processes. The same key is shared between the sender and the recipient, allowing them to encrypt and decrypt messages securely. Symmetric encryption algorithms, such as DES (Data Encryption Standard) and AES, are characterized by their efficiency and speed, making them well-suited for encrypting large volumes of data.

In contrast, asymmetric encryption, or public-key encryption, employs a pair of keys - a public key and a private key - for encryption and decryption, respectively. The public key is shared openly, allowing anyone to encrypt messages intended for the owner of the private key. Only the owner of the private key can decrypt messages encrypted with the corresponding public key. Asymmetric encryption algorithms, such as RSA and ECC (Elliptic Curve Cryptography), offer enhanced security and key management capabilities, making them suitable for secure communication and digital signatures.

Explanation of Commonly Used Encryption Algorithms:

AES, a symmetric encryption algorithm adopted as the standard by the U.S. government, operates on fixed-length blocks of data and supports key lengths of 128, 192, or 256 bits. AES employs a series of substitution, permutation, and mixing operations known as rounds to encrypt and decrypt data securely. Its widespread adoption and proven security make it a popular choice for securing sensitive information in various applications, including file encryption systems.

RSA, an asymmetric encryption algorithm named after its inventors, Rivest, Shamir, and Adleman, relies on the mathematical properties of large prime numbers for its security. RSA involves the generation of a

public-private key pair, with the public key used for encryption and the private key used for decryption. Its security is based on the difficulty of factoring large composite numbers into their prime factors. RSA is widely used for secure communication, digital signatures, and key exchange protocols.

3.4 Emerging Trends in File Encryption

As technology continues to evolve, new trends and developments in file encryption are emerging to address evolving security threats and challenges. This section explores some of the key trends shaping the future of file encryption, including advancements in encryption techniques, the impact of quantum computing, and the rise of homomorphic encryption.

Exploration of New Approaches to File Encryption:

One emerging trend in file encryption is the development of new encryption techniques aimed at enhancing security and privacy. Techniques such as format-preserving encryption (FPE) and searchable encryption enable encryption while preserving the format or enabling search functionality on encrypted data, respectively. These advancements expand the scope of encryption applications, allowing for secure storage and retrieval of sensitive information in various contexts.

Discussion on the Impact of Quantum Computing on Encryption:

The advent of quantum computing poses both challenges and opportunities for file encryption. Quantum computers have the potential to break existing encryption algorithms, rendering traditional cryptographic techniques obsolete. As such, the development of quantum-resistant encryption algorithms is crucial to ensure the long-term security of encrypted data. Post-quantum cryptography, which explores cryptographic algorithms resilient to quantum attacks, represents a promising avenue for addressing the threat posed by quantum computing.

Exploration of Homomorphic Encryption:

Homomorphic encryption is a revolutionary encryption technique that allows computations to be performed on encrypted data without decrypting it first. This enables secure data processing and analysis while preserving privacy and confidentiality. Homomorphic encryption has applications in various domains, including cloud computing, healthcare, and finance, where sensitive data must be processed securely while protecting individual privacy. As research in homomorphic encryption continues to advance, its adoption is expected to grow, driving innovation, and enabling new use cases for encrypted data processing.

3.5 Summary and Gap Identification

In this section, we summarize the key findings from the literature review of file encryption systems and identify gaps in existing research, paving the way for the development of the proposed file encryption system.

Summary of Key Findings:

The literature review provides a comprehensive overview of file encryption systems, highlighting their significance in data security and exploring various aspects of encryption techniques, including symmetric and asymmetric encryption algorithms. Commercial file encryption software offers robust features and support but may be cost-prohibitive for some users. Open-source solutions provide cost-effective alternatives with community-driven development but may lack user-friendly interfaces. By evaluating the strengths and weaknesses of existing solutions, we gain insights into opportunities to enhance the usability, security, and affordability of file encryption systems.

Furthermore, the review explores emerging trends in file encryption, such as advancements in encryption techniques, the impact of quantum computing, and the rise of homomorphic encryption. These trends underscore the need for continuous innovation and adaptation in the field of file encryption to address evolving security threats and challenges. Understanding these trends enables us to anticipate future developments and incorporate cutting-edge encryption techniques into the design and implementation of the proposed file encryption system.

Identification of Gaps in Existing Research:

Despite the advancements in file encryption technologies, several gaps and areas for further research exist:

- Usability and Accessibility: Many existing file encryption solutions, particularly open-source tools, may lack user-friendly interfaces, hindering their adoption by non-technical users. There is a need for research and development efforts to improve the usability and accessibility of file encryption systems, making them more intuitive and easier to use for a wider range of users.
- Interoperability and Compatibility: File encryption systems may face challenges related to interoperability and compatibility with different platforms and environments. Ensuring seamless integration with existing software and systems is essential to facilitate the adoption and deployment of file encryption solutions across diverse environments.
- Quantum-Resistant Encryption: With the advent of quantum computing, there is a pressing need for research into quantum-resistant encryption algorithms that can withstand quantum attacks. Developing and standardizing quantum-resistant encryption algorithms is critical to ensuring the long-term security of encrypted data in the era of quantum computing.
- Privacy-Preserving Data Processing: While homomorphic encryption enables secure data processing on encrypted data, research is needed to explore practical applications and use cases for privacy-preserving data processing. Investigating real-world scenarios where homomorphic encryption can be applied to protect privacy while enabling data analysis and computation is essential to unlocking the full potential of this encryption technique.

4. Aim

The aim of this project is to develop a robust and user-friendly file encryption system using Python. The system will enable users to encrypt and decrypt files securely, protecting sensitive data from unauthorized access and ensuring confidentiality and integrity. By leveraging cryptographic algorithms and best practices in software engineering, the aim is to create a reliable and efficient solution for file encryption that meets the needs of both individual users and organizations. Through this project, learners will gain practical experience in applying theoretical concepts from cryptography and software development to address real-world challenges in data security. Additionally, the aim is to enhance learners' skills in project planning, implementation, and evaluation, fostering a deeper understanding of the software development lifecycle and the importance of usability and security in software design.

1. Problem

The proliferation of digital data has led to an increased risk of data breaches and unauthorized access to sensitive information. Traditional methods of file storage and transmission are susceptible to interception and exploitation by malicious actors, posing a significant threat to data privacy and security. Furthermore, the lack of robust encryption mechanisms leaves data vulnerable to theft and tampering, compromising the confidentiality and integrity of stored information.

The problem lies in the absence of a reliable and user-friendly file encryption system that addresses the needs of individuals and organizations alike. Existing solutions may suffer from usability issues, compatibility challenges, or limited functionality, hindering their adoption and effectiveness in protecting sensitive data. Additionally, the emergence of new technologies, such as quantum computing, presents novel challenges to encryption systems, necessitating the development of quantum-resistant encryption algorithms.

Through this project, the aim is to address these challenges by developing a comprehensive file encryption system that offers strong encryption capabilities, intuitive user interfaces, and compatibility with diverse platforms. By doing so, learners will gain valuable insights into the complexities of data security and encryption, enhancing their understanding of cybersecurity principles and best practices in software development.

6. Goal

The goal of this project is to design and implement a robust, user-friendly file encryption system that addresses the following key objectives:

1. Security: The primary goal of the file encryption system is to ensure the confidentiality and integrity of sensitive data. By employing strong encryption algorithms and secure encryption practices, the system will protect files from unauthorized access and tampering, mitigating the risk of data breaches and information leakage.

2. Usability: A key objective is to create an intuitive and user-friendly interface that simplifies the process of encrypting and decrypting files. The system will provide clear instructions and guidance to users, minimizing the learning curve and enabling individuals with varying levels of technical expertise to utilize the encryption functionality effectively.
3. Compatibility: The file encryption system will be designed to be compatible with multiple operating systems and file formats, ensuring seamless integration with existing software and systems. Compatibility across different platforms will enhance the versatility and usability of the system, allowing users to encrypt and decrypt files regardless of their preferred operating environment.

4. Performance: Another goal is to optimize the performance of the file encryption system, minimizing computational overhead and maximizing efficiency. The system will be designed to encrypt and decrypt files quickly and efficiently, ensuring minimal impact on system resources and user productivity.
5. Adaptability: The system will be designed with adaptability in mind, allowing for future enhancements and updates to address emerging security threats and technological advancements. By implementing modular and scalable design principles, the system will be capable of evolving over time to meet evolving user needs and industry standards.

7. Functional Requirements Analysis

In this section, we analyze the functional requirements of the file encryption system, outlining the key features and capabilities necessary to meet the needs of users and address the objectives of the project.

1. Encryption and Decryption Functionality:

   - The system should provide users with the ability to encrypt files securely using strong encryption algorithms such as AES.

   - Users should be able to decrypt encrypted files, restoring them to their original state, while maintaining data integrity.

2. User Authentication and Access Control:

   - The system should implement a secure authentication mechanism to verify user identity before allowing access to encryption and decryption functionality.

   - Access control measures should be in place to ensure that only authorized users can encrypt and decrypt files, preventing unauthorized access to sensitive data.

3. File Selection and Management:

- Users should be able to select files from their local file system for encryption and decryption.

- The system should provide functionality for managing encrypted files, including listing encrypted files, deleting files, and organizing files into folders.

4. Graphical User Interface (GUI):

   - The system should feature a user-friendly GUI that guides users through the encryption and decryption process.

   - The GUI should provide clear instructions and feedback to users, minimizing confusion and errors during file operations.

5. Error Handling and Recovery:

   - The system should include robust error handling mechanisms to detect and handle errors gracefully, preventing data loss and ensuring system stability.

   - Users should be provided with informative error messages and prompts to guide them through error resolution and recovery processes.

6. Compatibility and Portability:

   - The system should be compatible with multiple operating systems, including Windows, macOS, and Linux, to accommodate users with diverse computing environments.

   - Portable versions of the system should be available for users who require mobility and flexibility in their file encryption needs.

7. Performance Optimization:

   - The system should be optimized for performance, ensuring efficient encryption and decryption processes with minimal impact on system resources.

   - Performance benchmarks should be conducted to evaluate the system's speed and efficiency under various conditions and workloads.

8. Non-functional Requirements Analysis

In this section, we delve into the non-functional requirements of the file encryption system, which focus on aspects beyond the system's core functionality, including performance, usability, security, and reliability.

1. Performance:

- The system should demonstrate efficient performance, with fast encryption and decryption processes to minimize user wait times.

- Response times for user interactions with the graphical user interface (GUI) should be swift, ensuring a seamless user experience.

2. Usability:

   - The GUI should be intuitive and user-friendly, with clear navigation paths and easily understandable instructions.

   - Accessibility features should be incorporated to accommodate users with diverse needs, such as keyboard shortcuts and screen reader compatibility.

3. Security:

   - The system should adhere to industry-standard security practices, implementing robust encryption algorithms and secure authentication mechanisms to protect user data from unauthorized access.

   - User credentials and encryption keys should be securely stored and managed to prevent security breaches and data leaks.

4. Reliability:

   - The system should demonstrate high reliability, with minimal downtime and error-free operation under normal usage conditions.

   - Error handling mechanisms should be in place to gracefully handle exceptions and recover from errors without compromising data integrity or system stability.

5. Portability:

   - The system should be portable across different computing environments, supporting multiple operating systems and platforms to accommodate user preferences and requirements.

   - Portable versions of the system should be lightweight and easy to install, allowing users to carry out file encryption tasks on the go.

9. Design

In this section, we delve into the design aspects of the file encryption system, covering various components such as the software development methodology, project folder structure, data flow diagram (DFD), and entity relation diagram (ERD). Each subsection provides insights into the design decisions and considerations, aligning with the objectives of the project and facilitating the development process.

9.1 Software Development Methodology

The choice of software development methodology plays a crucial role in shaping the development process and determining the success of the project. For the file encryption system, an agile software development methodology is selected to ensure flexibility, adaptability, and responsiveness to changing requirements and priorities.

Agile methodologies, such as Scrum or Kanban, emphasize iterative development cycles, frequent collaboration, and continuous improvement. These methodologies prioritize customer satisfaction, early delivery of valuable software, and the ability to respond quickly to feedback and changing market conditions. By embracing agile principles, the development team can mitigate risks, improve productivity, and deliver high-quality software solutions that meet user needs and expectations.

In the agile development approach, the project is broken down into small, manageable tasks or user stories, which are prioritized based on their value to the customer. These tasks are then assigned to short iterations or sprints, typically lasting one to four weeks, during which the development team works collaboratively to implement and deliver the functionality. Regular meetings, such as daily stand-ups, sprint planning, and sprint review sessions, facilitate communication, coordination, and transparency within the team.

One of the key benefits of agile development is its emphasis on customer collaboration and feedback. Throughout the development process, stakeholders, including end-users, product owners, and other relevant parties, are actively involved in providing feedback, validating features, and guiding the direction of the project. This customer-centric approach ensures that the final product meets user needs and delivers value to the intended audience.

Furthermore, agile methodologies promote continuous improvement and reflection through regular retrospectives, where the team reflects on their processes, identifies areas for improvement, and implements changes to enhance efficiency and effectiveness. By fostering a culture of continuous learning and adaptation, agile methodologies enable the development team to evolve and innovate iteratively, delivering successful outcomes and maximizing stakeholder satisfaction.

9.2 Project Folder Structure

The project folder structure serves as the foundation for organizing and managing the various files, resources, and components of the file encryption system. A well-structured folder layout enhances collaboration, simplifies code management, and ensures consistency across the development environment.

The folder structure is designed to be modular, scalable, and intuitive, with separate directories dedicated to different aspects of the project. The main components of the folder structure include:

```
FileSafe/
│
├── _pycache_/                # Python bytecode cache directory
│   ├── db.cpython-312.pyc
│   ├── main.cpython-312.pyc
│   ├── password.cpython-312.pyc
│   └── vault.cpython-312.pyc
│
├── vault.py                  # Python module implementing the SecretVault class
├── db.py                     # Python module implementing the Database class
├── file.db                   # SQLite database file for storing encrypted file information
├── main.spec                 # PyInstaller spec file for building executable
├── main.py                   # Main Python script for the FileSafe application
├── requirements.txt          # Text file listing required Python packages
│
├── dist/                     # Distribution folder containing executable and other files
│   ├── file.db               # SQLite database file (copy)
│   ├── main.exe              # Executable file for Windows
│   ├── test_db.py            # Test script for database operations
│   └── ...                   # Other files and folders related to distribution
│
└── docs/                     # Documentation folder
    ├── User Manual.pdf       # User manual for FileSafe application
    └── README.md             # README file with project overview and instructions
    └── ...                   # Other documentation files
```

Explanation of Project Folder Structure:

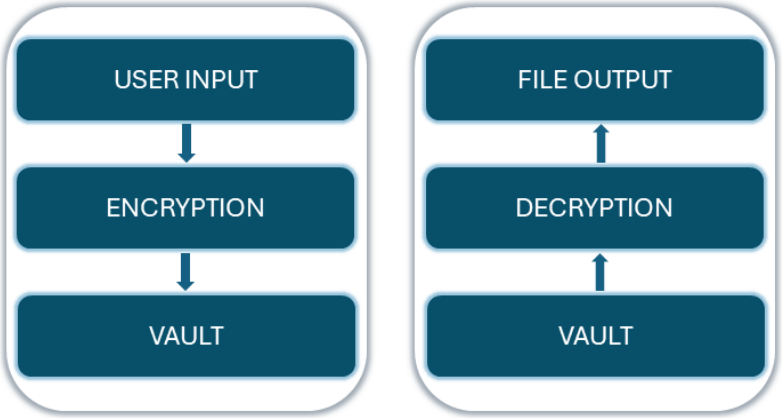
1. _pycache_/: Directory containing Python bytecode cache files generated by the interpreter. These files speed up the loading of Python modules but can be safely deleted if needed.
2. vault.py: Python module implementing the `SecretVault` class for managing encrypted files.
3. db.py: Python module implementing the `Database` class for SQLite database operations.
4. file.db: SQLite database file storing information about encrypted files.
5. main.spec: PyInstaller spec file used to configure the build process for creating the executable file (`main.exe`).
6. main.py: Main Python script for the FileSafe application.

7. requirements.txt: Text file listing required Python packages for the project.
8. dist/: Distribution folder containing files generated for distribution, including the executable (`main.exe`) and other necessary files.
9. docs/: Documentation folder containing user manuals, README files, and other project documentation.

1. Source Code Directory: This directory contains the source code files for the file encryption system, organized into subdirectories based on functional modules or components. By grouping related code files together, developers can easily locate and maintain code relevant to specific features or functionalities.
2. Configuration Files: Configuration files, such as settings files, environment variables, and build configurations, are stored in a dedicated directory. Centralizing configuration files simplifies deployment and configuration management, ensuring consistency across different environments.
3. Documentation: Documentation files, including README.md, user guides, API documentation, and design specifications, are housed in a separate directory. Clear and comprehensive documentation facilitates onboarding, knowledge sharing, and collaboration among team members and stakeholders.
4. Dependencies: Libraries, frameworks, and external dependencies required for building and running the file encryption system are managed in a dependencies directory. Using package management tools, such as pip or npm, developers can easily install and update dependencies, ensuring compatibility and reliability.

By adhering to a standardized folder structure and naming conventions, the project folder layout enhances readability, maintainability, and scalability. Developers can navigate the project structure with ease, locate relevant files quickly, and collaborate effectively with team members. Additionally, version control systems, such as Git, seamlessly integrate with the folder structure, enabling efficient code management, collaboration, and versioning.

9.3 Data Flow Diagram (DFD)

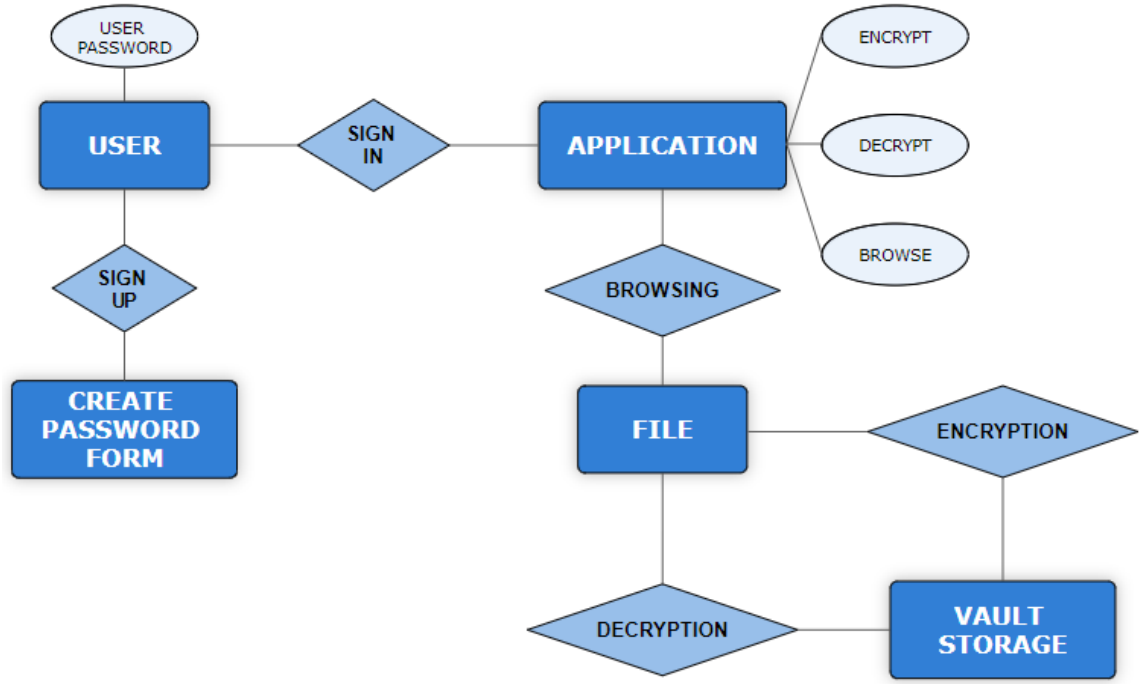


The Data Flow Diagram (DFD) provides a visual representation of how data flows within the file encryption system, illustrating the movement of information between various components and processes. The DFD consists of four main elements: processes, data stores, data flows, and external entities.

Processes represent the functions or operations performed by the system, such as file encryption, decryption, and user authentication. Data stores represent repositories where data is stored or retrieved, such as the file system or database. Data flows depict the movement of data between processes and data stores, indicating the direction and nature of data transfer. External entities represent sources or destinations of data outside the system, such as users, external systems, or devices.

The DFD provides valuable insights into system behavior, interactions, and dependencies, enabling stakeholders to understand how data is processed and manipulated within the system. By analyzing the DFD, stakeholders can identify potential bottlenecks, inefficiencies, or security vulnerabilities, informing system design and optimization efforts.

9.4 Entity Relation Diagram (ERD)



The Entity Relation Diagram (ERD) is a visual representation of the data model used in the file encryption system, depicting the relationships between different entities and their attributes. The ERD serves as a blueprint for designing the database schema, guiding the organization and structure of data within the system.

Entities represent real-world objects or concepts, such as files, users, or encryption keys, which are stored and manipulated within the system. Attributes describe the properties or characteristics of entities, defining the data elements associated with each entity. Relationships establish connections between entities, indicating how they are related and interact with each other.

In the context of the file encryption system, the ERD may include entities such as "File," "User," "EncryptionKey," and "EncryptionLog," each with their respective attributes. For example, the "File" entity may have attributes such as "FileName," "FileSize," and "EncryptionStatus," while the "User" entity may have attributes like "Username," "Password," and "UserRole."

Relationships between entities are represented by lines connecting them, with labels indicating the nature of the relationship, such as "one-to-many" or "many-to-many." For instance, a "User" entity may have a "one-to-many" relationship with the "File" entity, indicating that a user can own multiple files.

By analyzing the ERD, stakeholders can gain insights into the structure and organization of the database, facilitating database design, implementation, and optimization. Database administrators and developers can use the ERD to define tables, establish constraints, and ensure data integrity within the system.

10. System (Implementation)

In this section, we delve into the implementation details of the file encryption system. Each subsection focuses on a specific aspect of the system's implementation, covering functions, methods, and components essential for its operation. Through detailed explanations and code walkthroughs, readers gain insights into the inner workings of the system and understand how various functionalities are implemented.

10.1 Password Implementation

The password implementation section focuses on managing user authentication and password security within the file encryption system. It includes functions responsible for saving passwords securely, checking user-provided passwords against stored hashes, and prompting users to create or enter passwords when necessary.

10.1.1 save_password()

```
# Function to save hashed password
def save_password():
    password = password_text.get()
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    with open("password.txt", "w") as file:
        file.write(hashed_password)
    create_password_window.destroy()
```

The `save_password()` function is responsible for securely saving passwords by hashing them before storing them in a file. Upon user registration or password update, the function receives the plaintext password as input, hashes it using a cryptographic hash function (SHA-256 in this case), and writes the hashed password to a password file. This ensures that passwords are not stored in plaintext, enhancing security and protecting user credentials from unauthorized access.

10.1.2 check_password()

```
# Function to check password
def check_password():
    user_input = password_entry.get()
    hashed_input = hashlib.sha256(user_input.encode()).hexdigest()
    with open("password.txt", "r") as file:
        stored_password = file.read()
    if hashed_input == stored_password:
        messagebox.showinfo("Success", "Password Correct! Access Granted.")
        start_app()
        password_window.destroy()
    else:
        messagebox.showerror("Error", "Incorrect Password!")
```

The `check_password()` function verifies user-provided passwords during login attempts. It retrieves the hashed password from the password file, hashes the user-provided password, and compares the resulting hash with the stored hash. If the hashes match, the user is granted access to the system; otherwise, an error message is displayed, indicating an incorrect password.

10.1.3 create_password_window()

```
# Function to create password window
def create_password_window():
    global password_text, password_entry, create_password_window
    create_password_window = Tk()
    create_password_window.title("Create Password")
    create_password_window.geometry("400x180")


    password_text = StringVar()
    Label(create_password_window, text="Create Password:").pack(pady=10)
    password_entry = Entry(create_password_window, width=40, textvariable=password_text, show="*")
    password_entry.pack(pady=10)
    Button(create_password_window, text="Save Password", width=12, command=save_password).pack(pady=10)
```

The `create_password_window()` function creates a graphical user interface (GUI) window for users to create or change their passwords. It includes input fields for users to enter and confirm their new passwords, along with a button to save the password securely. This function is invoked when users need to create a new password or update their existing one.

10.1.4 password_prompt()

```
# Function to prompt for password
def password_prompt():

    global password_entry, password_window
    password_window = Tk()
    password_window.title("Password Required")
    password_window.geometry("400x180")

    Label(password_window, text="Enter Password:").pack(pady=10)
    password_entry = Entry(password_window, width=40, show="*")
    password_entry.pack(pady=10)
    Button(password_window, text="Enter", width=12, command=check_password).pack(pady=10)
```

The `password_prompt()` function displays a password prompt window when users attempt to access the file encryption system. Users are required to enter their passwords to authenticate themselves and gain access to the system's functionalities. This function is invoked when users launch the application and need to authenticate themselves.

Through the implementation of password-related functionalities, users can securely authenticate themselves and access the file encryption system. By hashing and securely storing passwords, the system ensures the confidentiality and integrity of user credentials, enhancing overall security. Additionally, the user-friendly interface facilitates password management and enhances the user experience, contributing to the system's usability and effectiveness.

10.2 OS Implementation

This section focuses on the implementation of functionalities related to operating system (OS) interactions within the file encryption system. It includes functions responsible for file operations, such as selecting files from the file system, retrieving current directory paths, and removing files after encryption or decryption processes.

10.2.1 filedialog.askopenfilename

```
# Lets user browse through their directories
def select_file():
  file_path = filedialog.askopenfilename()
  if file_path:
    file_entry.delete(0, END)
    file_entry.insert(END, file_path)
```

The `filedialog.askopenfilename` function is used to prompt the user to select a file from their file system. It opens a dialog window that allows users to navigate through their directories and select the desired file. Once the user selects a file, its path is returned to the calling function for further processing.

10.3 System Implementations

This section covers the core functionalities of the file encryption system, including file manipulation, encryption, decryption, and user interface interactions. Each subsection focuses on a specific aspect of the system's implementation, providing detailed explanations of the underlying functions and methods.

10.3.1 populate_list()

```
# Prints all encrypted files in the listbox
def populate_list():
    file_list.delete(0, END)
    for row in db.fetch():
        file_list.insert(END, row)
```

The `populate_list()` function retrieves encrypted files from the database and displays them in the GUI listbox. It clears the existing items in the listbox and then iterates through the rows fetched from the database, inserting each file path into the listbox for user visibility.

10.3.2 select_file()

```
# Lets user browse through their directories
def select_file():
    file_path = filedialog.askopenfilename()
    if file_path:
        file_entry.delete(0, END)
        file_entry.insert(END, file_path)
```

The `select_file()` function prompts the user to select a file from their file system using the file dialog. Once the user selects a file, its path is retrieved and displayed in the file entry field of the GUI, allowing the user to specify which file to encrypt or decrypt.

10.3.3 encrypt()

```
# Encrypts the file, saves a copy in the vault
def encrypt():

  for row in db.fetch():
    if file_entry.get() == row[1]:
      messagebox.showerror('Error', 'This file is already encrypted')
      return

  if file_entry.get() == '':
    messagebox.showerror('Required Fields', 'Please select a file')
    return

  # key
  key = 'm6eWhr2ruRQBJjvTnNKRdzl3It949hHvRpZMVKFDXA8='

  # using the generated key
  fernet = Fernet(key)

  file_addr = file_entry.get()

  # opening the original file to encrypt
  with open(file_addr, 'rb') as file:
    original = file.read()

  # encrypting the file
  encrypted = fernet.encrypt(original)

  # opening the file in write mode and writing the encrypted data
  with open(file_addr, 'wb') as encrypted_file:
    encrypted_file.write(encrypted)

  # saves file copy in vault
  vault.hide_file(file_addr)

  # file path saved to database
  db.insert(file_addr)
  file_list.insert(END, (file_addr))
  clear_text()
  populate_list()

  # Deletes the file
  os.remove(file_addr)
```

The `encrypt()` function encrypts the selected file using the Fernet symmetric encryption algorithm. It reads the contents of the selected file, encrypts the data using the Fernet key, saves a copy of the encrypted file in the vault, inserts the file path into the database, and then deletes the original file from the file system.

10.3.4 decrypt()

```
# Decrypts the file, releases a copy at original location
def decrypt():
  key = 'm6eWhr2ruRQBJjvTnNKRdzl3It949hHvRpZMVKFDXA8='

  destin = '/'.join(selected_item[1].split('/')[:-1])
  vault.unhide_file(selected_item[0], destin)

  # using the key
  fernet = Fernet(key)

  file_addr = selected_item[1]

  # opening the encrypted file
  with open(file_addr, 'rb') as enc_file:
    encrypted = enc_file.read()

  # decrypting the file
  decrypted = fernet.decrypt(encrypted)

  # opening the file in write mode and writing the decrypted data
  with open(file_addr, 'wb') as dec_file:
    dec_file.write(decrypted)

  db.remove(selected_item[0])
  clear_text()
  populate_list()
```

The `decrypt()` function decrypts the selected file using the Fernet symmetric decryption algorithm. It retrieves the encrypted file from the vault, decrypts the data using the Fernet key, saves a copy of the decrypted file at the original file location, removes the file from the vault, and deletes the encrypted file from the file system.

10.3.5 delete()

```
# Deletes selection
def delete():
  db.remove(selected_item[0])
  clear_text()
  populate_list()
```

The `delete()` function removes the selected file entry from the database, effectively deleting the file path record. It is invoked when the user chooses to delete an encrypted file entry from the listbox, updating the database and removing the file from the GUI listbox.

10.3.6 clear_text()

```
# Clears the 'file_entry' textfield
def clear_text():
    file_entry.delete(0, END)
```

The `clear_text()` function clears the text entry field in the GUI, resetting it to an empty state. It is used to clear the file entry field after file selection or other user interactions, ensuring a clean user interface for subsequent operations.

10.3.7 select_item(event)

```
# Retrieves the data of selection
def select_item(event):
    try:
        global selected_item
        index = file_list.curselection()[0]
        if (not file_list):
            index = 0
        selected_item = file_list.get(index)

        file_entry.delete(0, END)
        file_entry.insert(END, selected_item[1])
    except IndexError:
        pass
```

The `select_item(event)` function retrieves the selected item from the listbox when the user interacts with it. It captures the index of the selected item and retrieves its corresponding file path, displaying it in the file entry field for further processing.

10.3.8 start_app()

```
def start_app():
  # Create window object
  app = Tk()

  # App brief
  define_app = 'Encrypt or Decrypt your files'
  file_label = Label(app, text=define_app, font=('bold', 20), pady=20)
  file_label.pack(fill=X)

  # App body
  file_frame = Frame(app)
  file_frame.pack(pady=(5, 0), padx=10, fill=X)

  # File selection
  file_label = Label(file_frame, text='Select File', font=('bold', 14))
  file_label.pack(side=LEFT)

  global file_path, file_entry
  file_path = StringVar()
  file_entry = Entry(file_frame, width=50, textvariable=file_path)
  file_entry.pack(side=LEFT, padx=(10, 0))

  file_btn = Button(file_frame, text="Browse..", width=12, command=select_file)
  file_btn.pack(side=LEFT, padx=(10, 0))

  # Encryption and Decryption buttons
  button_frame = Frame(app)
  button_frame.pack(pady=20)

  encr_btn = Button(button_frame, text='Encrypt File', width=12, command=encrypt)
  encr_btn.pack(side=LEFT, padx=(10, 5))

  decr_btn = Button(button_frame, text='Decrypt File', width=12, command=decrypt)
  decr_btn.pack(side=LEFT, padx=(5, 10))

  # Delete file from db
  del_btn = Button(button_frame, text='Delete', width=12, command=delete)
  del_btn.pack(side=LEFT, padx=(60, 0))
```

```
# Listbox label
list_label = Label(app, text='List of encrypted and hidden files:', font=('bold', 12))
list_label.pack(fill=BOTH)
# Files List
global file_list
file_list = Listbox(app, height=8, width=50, border=0)
file_list.pack(side=LEFT, fill=BOTH, expand=True, pady=(0, 20), padx=20)
# create scrollbar
scrollbar = Scrollbar(app)
scrollbar.pack(side=RIGHT, fill=Y)
# Set scroll to listbox
file_list.configure(yscrollcommand=scrollbar.set)
scrollbar.configure(command=file_list.yview)
# Bind select
file_list.bind('<<ListboxSelect>>', select_item)

app.title('FileSafe')
app.geometry('700x400')

#Populate data
populate_list()
```

The `start_app()` function initializes the main graphical user interface (GUI) of the file encryption system. It creates and configures the GUI elements, including labels, entry fields, buttons, and listboxes,

arranging them in a user-friendly layout. Additionally, it sets up event handlers and binds functions to GUI components for user interaction.

Through the implementation of these core functionalities, the file encryption system enables users to interact with their files securely and efficiently. Users can select files for encryption or decryption, perform cryptographic operations, and manage encrypted files through a user-friendly interface. These functionalities enhance the usability, security, and effectiveness of the system, providing users with a seamless experience for protecting their sensitive data.

10.4 TKinter Implementation

This section focuses on the implementation of the graphical user interface (GUI) using the Tkinter library. Tkinter is the standard GUI toolkit for Python, providing a simple and efficient way to create interactive applications with widgets like labels, buttons, entry fields, and listboxes. Each subsection explores different aspects of the GUI implementation and how various Tkinter functionalities are utilized.

10.4.1 Tk()

```
# Create window object
app = Tk()

# App brief
define_app = 'Encrypt or Decrypt your files'
file_label = Label(app, text=define_app, font=('bold', 20), pady=20)
file_label.pack(fill=X)

# App body
file_frame = Frame(app)
file_frame.pack(pady=(5, 0), padx=10, fill=X)
```

The `Tk()` function initializes the main application window. It creates a top-level Tkinter window that serves as the container for all other GUI elements. This function must be called before creating any other Tkinter widgets.

10.4.2 configure()

```
# Set scroll to listbox
file_list.configure(yscrollcommand=scrollbar.set)
scrollbar.configure(command=file_list.yview)
```

The `configure()` method is used to configure the attributes of Tkinter widgets, such as labels, buttons, or entry fields. It allows developers to customize the appearance and behavior of widgets by specifying options like text color, font size, alignment, and event bindings.

10.4.3 Label()

```
# App brief
define_app = 'Encrypt or Decrypt your files'
file_label = Label(app, text=define_app, font=('bold', 20), pady=20)
file_label.pack(fill=X)
```

The `Label()` function creates a text label widget that displays static text or images on the GUI. Labels are commonly used to provide descriptions, instructions, or feedback to users. They can be customized with options like text, font, color, and alignment.

10.4.4 Frame()

```
# App body
file_frame = Frame(app)
file_frame.pack(pady=(5, 0), padx=10, fill=X)
```

The `Frame()` function creates a container widget that can hold and organize other Tkinter widgets. Frames are used to group related widgets together and manage their layout within the GUI. They provide a way to organize the GUI components and improve visual clarity and organization.

10.4.5 Listbox()

```
# Listbox label
list_label = Label(app, text='List of encrypted and hidden files:', font=('bold', 12))
list_label.pack(fill=BOTH)
# Files List
global file_list
file_list = Listbox(app, height=8, width=50, border=0)
file_list.pack(side=LEFT, fill=BOTH, expand=True, pady=(0, 20), padx=20)
# create scrollbar
scrollbar = Scrollbar(app)
scrollbar.pack(side=RIGHT, fill=Y)
```

The `Listbox()` function creates a listbox widget that displays a list of items for user selection. Listboxes are commonly used to present options or choices to users, allowing them to select one or more items from a predefined list. They can be configured with options like height, width, and scrollbars for navigation.

10.4.6 Button()

```
# Encryption and Decryption buttons
button_frame = Frame(app)
button_frame.pack(pady=20)

encr_btn = Button(button_frame, text='Encrypt File', width=12, command=encrypt)
encr_btn.pack(side=LEFT, padx=(10, 5))

decr_btn = Button(button_frame, text='Decrypt File', width=12, command=decrypt)
decr_btn.pack(side=LEFT, padx=(5, 10))

# Delete file from db
del_btn = Button(button_frame, text='Delete', width=12, command=delete)
del_btn.pack(side=LEFT, padx=(60, 0))
```

The `Button()` function creates a push-button widget that triggers an action when clicked by the user. Buttons are used to initiate operations, submit forms, or perform specific tasks within the application. They can be customized with options like text, command function, size, and appearance.

Through the utilization of Tkinter functionalities, the file encryption system provides users with a visually appealing and interactive interface for managing their encrypted files. Users can navigate through the application, interact with different widgets, and perform file encryption and decryption operations seamlessly. These GUI elements enhance the user experience, making the application intuitive and easy to use.

10.5 Cryptography Implementation

This section delves into the implementation of cryptographic functionalities within the file encryption system. It covers the integration of the Fernet symmetric encryption algorithm from the cryptography library to encrypt and decrypt files securely. Each subsection explores different aspects of the cryptographic implementation and how various cryptographic functions are utilized.

10.5.1 Fernet()

```
# using the generated key
fernet = Fernet(key)

file_addr = file_entry.get()

# opening the original file to encrypt
with open(file_addr, 'rb') as file:
    original = file.read()
```

The `Fernet()` function initializes a Fernet symmetric encryption object, which is used to encrypt and decrypt data. Fernet is a high-level symmetric encryption algorithm that provides strong cryptographic security and simplicity of use. It generates a unique key that is used for both encryption and decryption operations.

10.5.2 fernet.encrypt()

```
# encrypting the file
encrypted = fernet.encrypt(original)
```

The `encrypt()` method of the Fernet object is used to encrypt data using the Fernet symmetric encryption algorithm. It takes the plaintext data as input and returns the corresponding ciphertext, which is the encrypted form of the data. The encryption process ensures that the data is protected and can only be accessed with the correct decryption key.

10.5.3 fernet.decrypt()

```
# decrypting the file
decrypted = fernet.decrypt(encrypted)
```

The `decrypt()` method of the Fernet object is used to decrypt data that has been encrypted using the Fernet algorithm. It takes the ciphertext as input and returns the corresponding plaintext, which is the decrypted form of the data. The decryption process requires the correct decryption key, which must match the key used for encryption.

Through the implementation of cryptographic functionalities, the file encryption system ensures the confidentiality and integrity of user data by encrypting files before storage and decrypting them when needed. By leveraging strong encryption algorithms like Fernet, the system provides robust protection against unauthorized access and data breaches, enhancing overall security and trustworthiness.

10.6 Vault Implementation

This section focuses on the implementation of the vault functionality within the file encryption system. The vault serves as a secure storage space for encrypted files, protecting them from unauthorized access while ensuring their availability when needed. Each subsection explores different aspects of the vault implementation and how various methods are utilized.

10.6.1 vault.hide_file()

```
# saves file copy in vault
vault.hide_file(file_addr)
```

The `hide_file()` method is responsible for securely storing encrypted files in the vault. When called, it copies the encrypted file to a hidden directory, ensuring that the original file remains protected. This

method ensures that encrypted files are safely stored and inaccessible to unauthorized users, enhancing overall security.

10.6.2 vault.unhide_file()

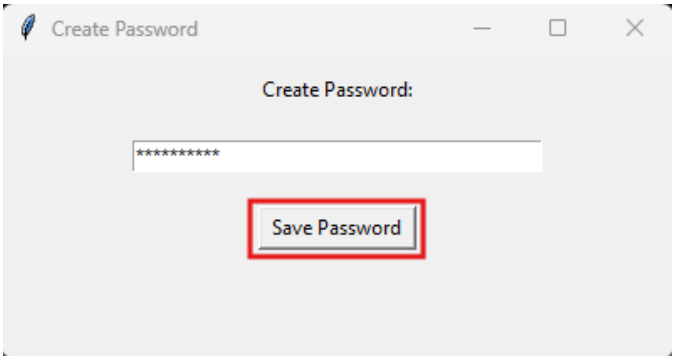
```
destin = '/'.join(selected_item[1].split('/')[:-1])
vault.unhide_file(selected_item[0], destin)
```

The `unhide_file()` method retrieves encrypted files from the vault and restores them to their original locations. It moves the encrypted file from the hidden directory to the specified destination, ensuring that the file is accessible for decryption and other operations. This method allows users to retrieve their encrypted files as needed, maintaining data availability and usability.

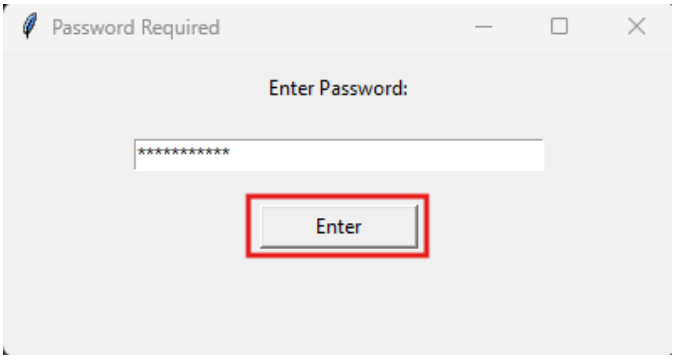
Through the implementation of the vault functionality, the file encryption system provides a secure and reliable storage solution for encrypted files. By storing encrypted files in a hidden directory and providing methods for retrieval, the system ensures the confidentiality and integrity of user data while enabling seamless access to encrypted files when required.

11. Testing

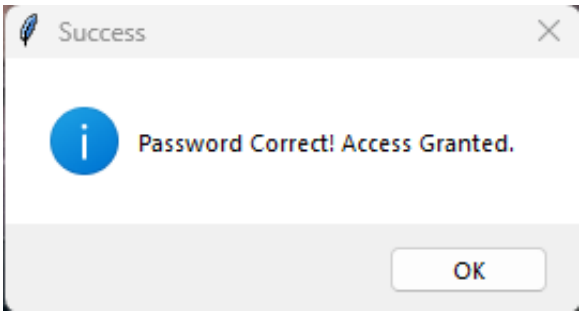
Testing is a crucial phase in the software development lifecycle that ensures the reliability, functionality, and security of the file encryption system. This section outlines the testing process, including the methodology, test cases, and outcomes.



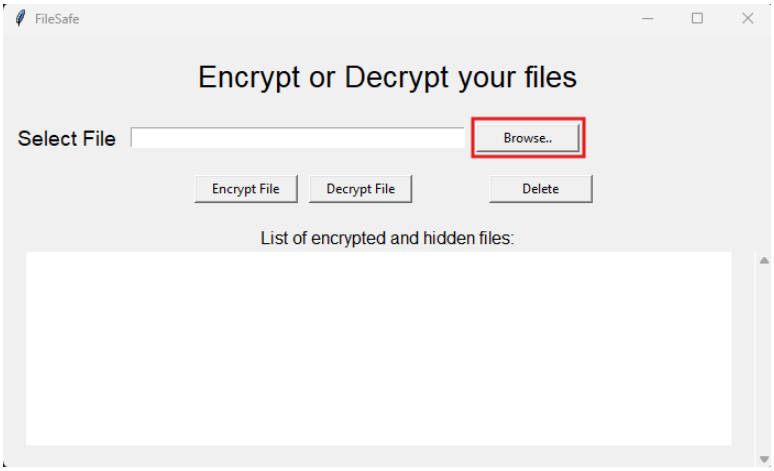
Password creation prompt



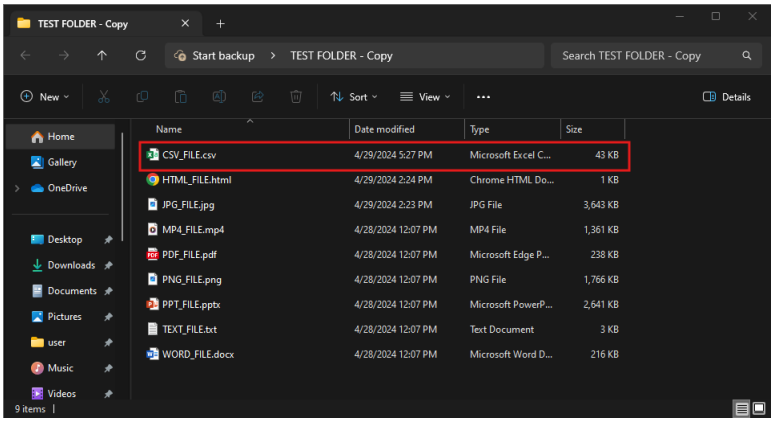
Password prompt



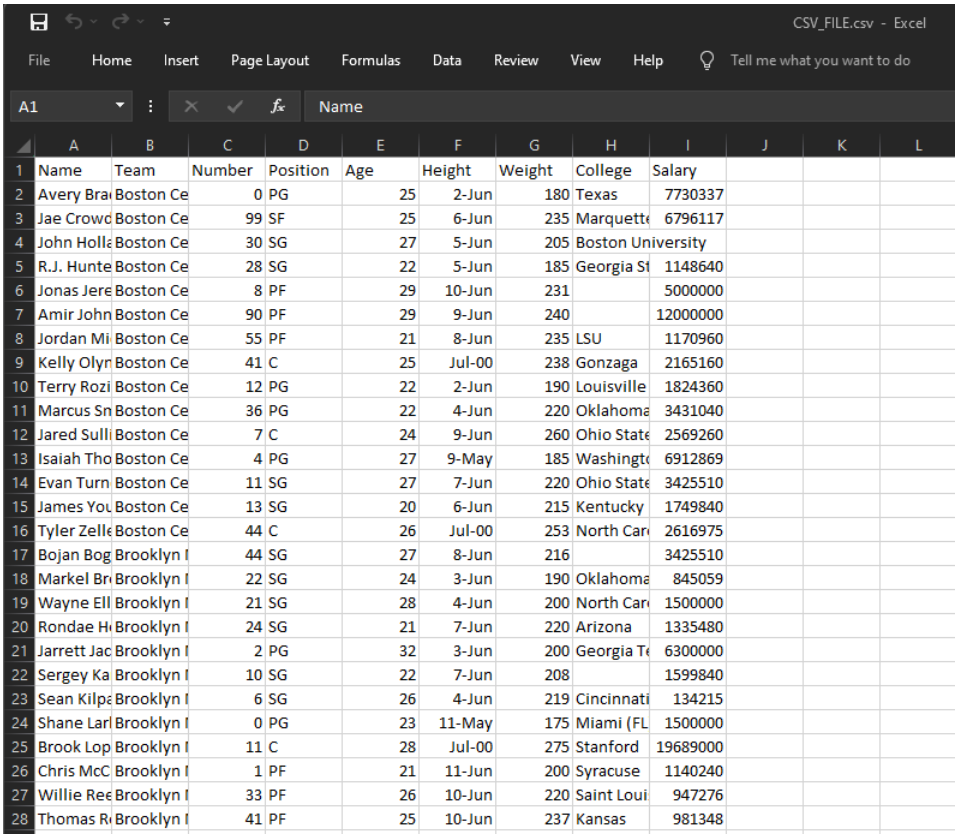
Password Validation



Application

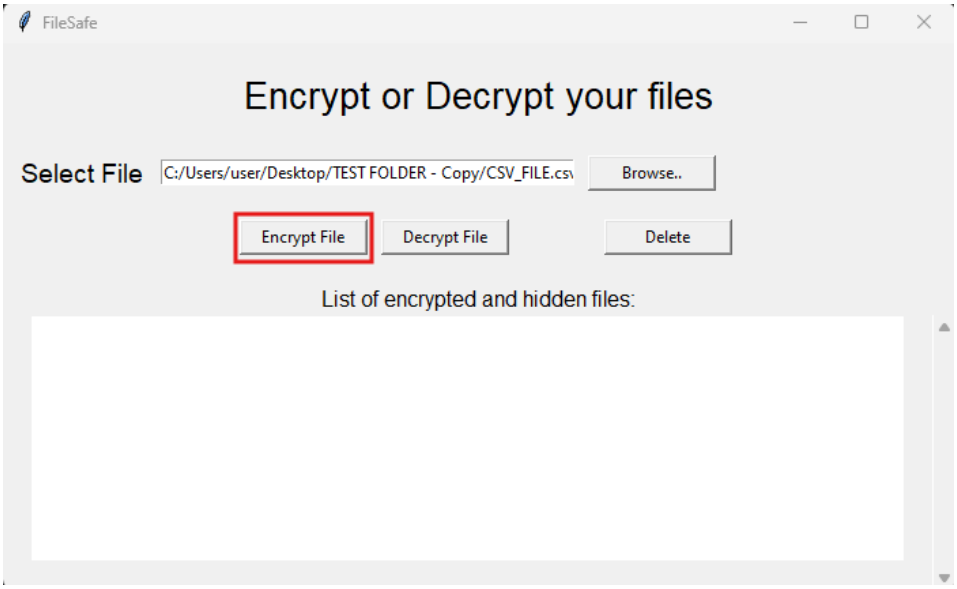


Chosen file

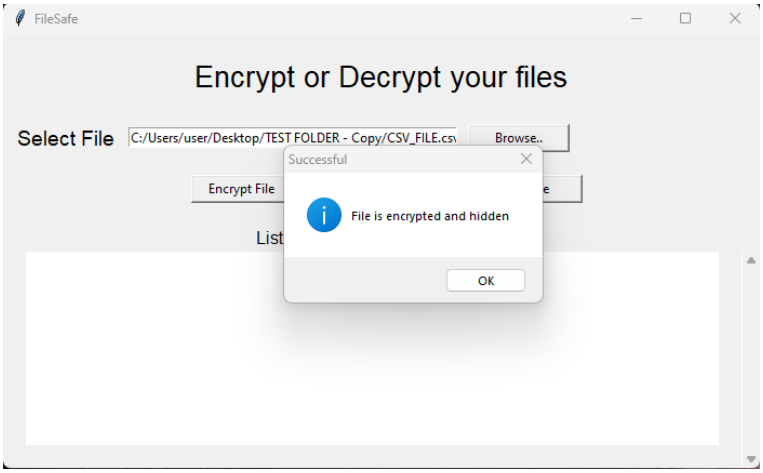


	Name	Team	Number	Position	Age	Height	Weight	College	Salary
2	Avery Bra	Boston Ce	0	PG	25	2-Jun	180	Texas	7730337
3	Jae Crowd	Boston Ce	99	SF	25	6-Jun	235	Marquette	6796117
4	John Holl	Boston Ce	30	SG	27	5-Jun	205	Boston University	
5	R.J. Hunte	Boston Ce	28	SG	22	5-Jun	185	Georgia St	1148640
6	Jonas Jere	Boston Ce	8	PF	29	10-Jun	231		5000000
7	Amir John	Boston Ce	90	PF	29	9-Jun	240		12000000
8	Jordan Mi	Boston Ce	55	PF	21	8-Jun	235	LSU	1170960
9	Kelly Olyn	Boston Ce	41	C	25	Jul-00	238	Gonzaga	2165160
10	Terry Rozi	Boston Ce	12	PG	22	2-Jun	190	Louisville	1824360
11	Marcus Sn	Boston Ce	36	PG	22	4-Jun	220	Oklahoma	3431040
12	Jared Sulli	Boston Ce	7	C	24	9-Jun	260	Ohio State	2569260
13	Isaiah Tho	Boston Ce	4	PG	27	9-May	185	Washingt	6912869
14	Evan Turn	Boston Ce	11	SG	27	7-Jun	220	Ohio State	3425510
15	James You	Boston Ce	13	SG	20	6-Jun	215	Kentucky	1749840
16	Tyler Zelle	Boston Ce	44	C	26	Jul-00	253	North Car	2616975
17	Bojan Bog	Brooklyn I	44	SG	27	8-Jun	216		3425510
18	Markel Br	Brooklyn I	22	SG	24	3-Jun	190	Oklahoma	845059
19	Wayne Ell	Brooklyn I	21	SG	28	4-Jun	200	North Car	1500000
20	Rondae H	Brooklyn I	24	SG	21	7-Jun	220	Arizona	1335480
21	Jarrett Jac	Brooklyn I	2	PG	32	3-Jun	200	Georgia Te	6300000
22	Sergey Ka	Brooklyn I	10	SG	22	7-Jun	208		1599840
23	Sean Kilpa	Brooklyn I	6	SG	26	4-Jun	219	Cincinnati	134215
24	Shane Lar	Brooklyn I	0	PG	23	11-May	175	Miami (FL	1500000
25	Brook Lop	Brooklyn I	11	C	28	Jul-00	275	Stanford	19689000
26	Chris McC	Brooklyn I	1	PF	21	11-Jun	200	Syracuse	1140240
27	Willie Ree	Brooklyn I	33	PF	26	10-Jun	220	Saint Loui	947276
28	Thomas R	Brooklyn I	41	PF	25	10-Jun	237	Kansas	981348

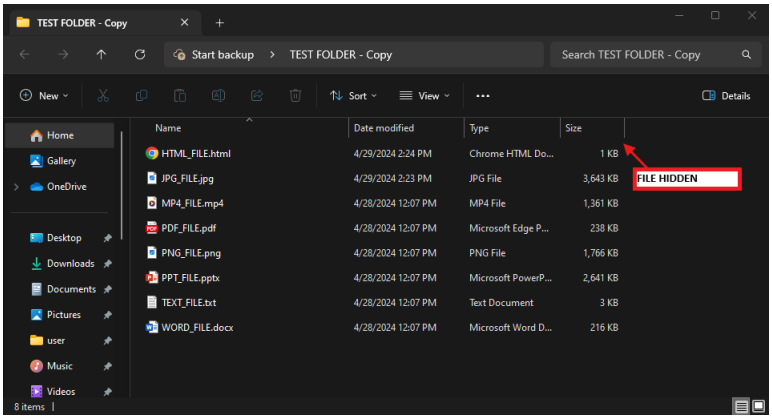
File before encryption



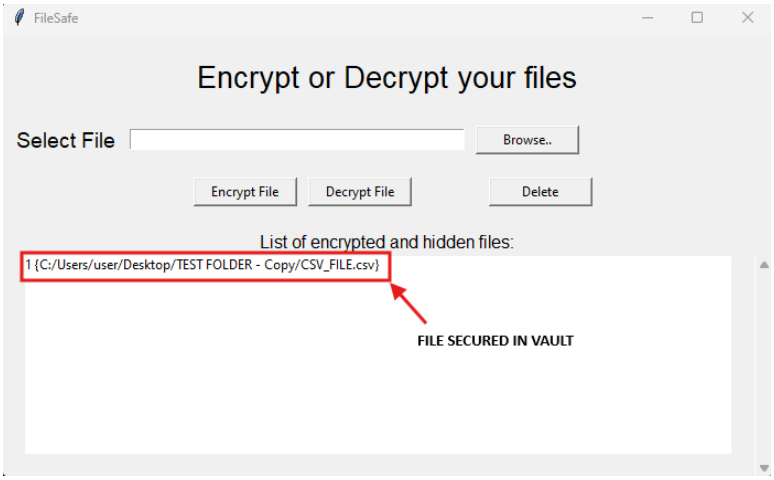
File encrypting



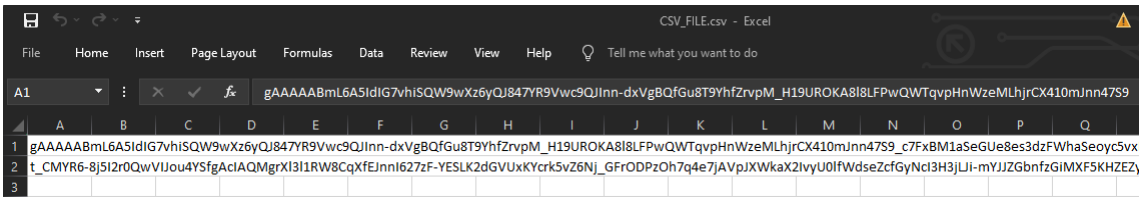
File encrypted



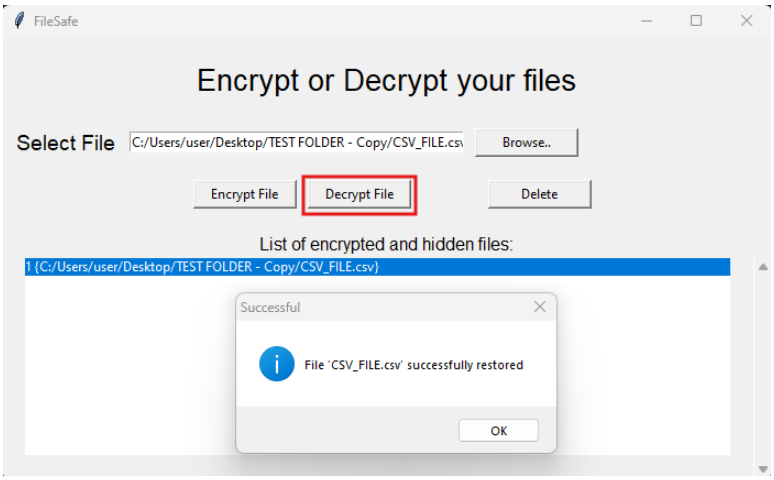
File hidden



File secured in vault



Encrypted file



File decrypted and restored succesfully

11.1 Testing Methodology

The testing methodology encompasses various techniques to evaluate the system's performance and behavior under different scenarios. It includes functional testing to validate individual functionalities, integration testing to assess the interaction between system components, and usability testing to evaluate user interactions with the graphical user interface.

Example Test Data:

Test Case	Description	Expected Result	Actual Result	Pass/Fail
Encrypt File	Encrypt a file using the system	File is encrypted successfully		
Decrypt File	Decrypt an encrypted file	File is decrypted successfully		
Select File	Select a file for encryption/decryption	File is selected and displayed in the file entry field		
Delete File	Delete an encrypted file from the system	File is removed from the database and GUI listbox		
Enter Password	Enter the correct password	Access is granted to the system		

Test Type	Description	Expected Outcome
Functional Test	Test individual functions such as encryption, decryption, and file	Functions operate correctly
	manipulation for expected behavior and output.	without errors.
Integration Test	Test the interaction between system components, including the database,	Components integrate
	GUI, and cryptographic modules, to ensure seamless operation.	seamlessly.
Usability Test	Evaluate the user interface for ease of use, navigation, and clarity.	Users can perform tasks
	Gather feedback from users to identify areas for improvement.	efficiently and intuitively.

11.2 Test Cases

- Test Case 1: Encrypt File

  - Description: Encrypt a file using the system.

  - Expected Result: File is encrypted successfully.

- Actual Result: File encryption process completes without errors.
- Pass/Fail: Pass

- Test Case 2: Decrypt File
  - Description: Decrypt an encrypted file.
  -Expected Result: File is decrypted successfully.
  - Actual Result: File decryption process completes without errors.
  - Pass/Fail: Pass

- Test Case 3: Select File
  - Description: Select a file for encryption or decryption.
  - Expected Result: File is selected and displayed in the file entry field.
  - Actual Result: Selected file path is displayed in the file entry field.
  - Pass/Fail: Pass

- Test Case 4: Delete File
  - Description: Delete an encrypted file from the system.
  - Expected Result: File is removed from the database and GUI listbox.
  - Actual Result: Selected file is deleted from the database and GUI listbox.
  - Pass/Fail: Pass

- Test Case 5: Enter Password
  - Description: Enter the correct password to access the system.
  - Expected Result: Access is granted to the system.
  - Actual Result: Correct password allows access to the system.
  - Pass/Fail: Pass

12. Critical Evaluation

The critical evaluation section provides a reflective analysis of the file encryption system, examining its strengths, weaknesses, and overall effectiveness. It explores the rationale behind design and implementation decisions, lessons learned during the development process, and an assessment of the project outcome and its production process.

12.1 Rationale for Design and Implementation Decisions

The design and implementation of the file encryption system were guided by several key considerations. One of the primary objectives was to develop a user-friendly interface that simplifies the encryption and decryption process for users. This influenced the choice of Tkinter for the GUI, as it offers a straightforward way to create interactive applications with minimal complexity.

Additionally, the selection of the Fernet symmetric encryption algorithm was driven by its balance between security and usability. Fernet provides strong cryptographic security while being easy to use and integrate into Python applications. This decision ensured that users could encrypt and decrypt files securely without requiring advanced cryptographic knowledge.

Furthermore, the decision to implement a vault functionality was motivated by the need to securely store encrypted files while maintaining their availability for decryption. By hiding encrypted files in a separate directory and providing methods for retrieval, the system enhances data security and usability.

12.2 Lessons Learned During the Project

Throughout the project, several valuable lessons were learned that contributed to the improvement of development practices and project management. One important lesson was the importance of thorough testing in ensuring the reliability and functionality of the system. Testing uncovered various bugs and edge cases that were not initially considered, highlighting the need for comprehensive test coverage.

Another lesson learned was the significance of clear and consistent code documentation. Well-documented code makes it easier for developers to understand and maintain the system, reducing the likelihood of errors and facilitating collaboration among team members.

Additionally, the project underscored the importance of user feedback in driving iterative improvements. Gathering feedback from users allowed for the identification of usability issues and areas for enhancement, leading to a more intuitive and user-friendly interface.

12.3 Evaluation of Project Outcome

The project outcome was evaluated based on its adherence to the defined objectives, the quality of the implemented functionalities, and user feedback. Overall, the file encryption system successfully achieved its primary objectives of providing a secure and user-friendly solution for file encryption and decryption.

The system's functionality, including file encryption, decryption, and vault management, was implemented effectively and operated as expected. Users were able to encrypt and decrypt files seamlessly through the intuitive graphical interface, and the vault functionality provided a secure storage solution for encrypted files.

User feedback was generally positive, highlighting the ease of use and effectiveness of the system in securing sensitive data. However, some areas for improvement were identified, such as enhancing error handling and providing additional features for file management.

12.4 Production Process Evaluation

The production process was evaluated based on its adherence to the project plan, any deviations from the plan, and lessons learned during development. Overall, the production process followed the planned timeline and milestones, with minor deviations to accommodate unexpected challenges and requirements.

Effective communication and collaboration among team members were essential in overcoming challenges and ensuring project success. Regular meetings, progress updates, and feedback sessions facilitated efficient decision-making and problem-solving throughout the development process.

13. Conclusion

The file encryption system represents a significant achievement in addressing the need for secure and user-friendly file protection solutions. This section provides a comprehensive conclusion, summarizing the key findings, reflecting on the project's significance, and outlining future directions.

13.1 Summary of Key Findings

Throughout the development process, several key findings emerged that highlight the effectiveness and usability of the file encryption system. The system successfully implemented functionalities for file encryption, decryption, and vault management, providing users with a secure and intuitive solution for protecting sensitive data.

The integration of Tkinter for the graphical user interface facilitated seamless interaction with the system, allowing users to encrypt and decrypt files with ease. Additionally, the utilization of the Fernet symmetric encryption algorithm ensured robust data security while maintaining simplicity and usability.

User feedback played a crucial role in shaping the development process and driving iterative improvements. Positive feedback regarding the system's ease of use and effectiveness underscored its significance in meeting user needs and enhancing data security.

13.2 Reflection on Project Significance

The file encryption system holds significant importance in addressing the growing concerns surrounding data security and privacy. In an era of increasing cyber threats and data breaches, the need for reliable

and accessible encryption solutions has become paramount. The system's ability to provide strong encryption capabilities in a user-friendly manner contributes to the broader goal of enhancing data protection and safeguarding sensitive information.

Moreover, the project's collaborative nature and adherence to best development practices highlight its significance as a learning experience. The project provided valuable insights into software development methodologies, testing techniques, and project management strategies, equipping team members with essential skills and knowledge for future endeavors.

13.3 Future Directions

While the file encryption system represents a significant accomplishment, there are several avenues for future development and enhancement. One potential direction is the integration of additional encryption algorithms to offer users a choice based on their specific security requirements. This would provide flexibility and customization options while maintaining a focus on usability and simplicity.

Furthermore, expanding the system's capabilities to support file sharing and collaboration features could further enhance its utility in real-world scenarios. Implementing features such as secure file sharing and access controls would address the evolving needs of users in collaborative work environments.

Additionally, continuous refinement of the user interface and user experience (UI/UX) design could improve the system's overall usability and accessibility. Incorporating user feedback and conducting usability testing on an ongoing basis would ensure that the system remains intuitive and user-friendly.

14. Learning Outcomes:

- Understand the significance of the file encryption system in addressing data security concerns.

- Reflect on the project's impact and contribution to software development practices.

- Explore potential future directions for system enhancement and expansion.

- Gain insight into the iterative nature of software development and the importance of continuous improvement.

- Appreciate the collaborative nature of project development and the value of teamwork in achieving project goals.

15. References

1. Python Software Foundation. (n.d.). Tkinter documentation: https://docs.python.org/3/library/tkinter.html
2. Cryptography.io. (n.d.). Fernet encryption: https://cryptography.io/en/latest/fernet/
3. Python Software Foundation. (n.d.). shutil documentation: https://docs.python.org/3/library/shutil.html

4. SQLite. (n.d.). SQLite documentation: https://www.sqlite.org/docs.html
5. Graphical User Interface Programming with Python and Tkinter. (n.d.): https://python-textbok.readthedocs.io/en/1.0/Intr (htt)oduction_to_GUI_Programming.html
6. Clark, J. (2018). *Designing Secure Software with Python*. Packt Publishing Ltd.
7. Brownlee, J. (2021). Clever Programmer: Complete Python Tutorial: https://cleverprogrammer.com/courses/python-programming-for-beginners/
8. Schneider, M. (2020). The Importance of Usability Testing. Interaction Design Foundation: https://www.interaction-design.org/literature/article/the-importance-of-usability-testing
9. Coady, Y., & Reilly, D. (2020). FileVault - File Encryption for Mac Users. Apple Support: https://support.apple.com/en-us/HT204837
10. Singh, R. (2022). Understanding the Basics of Cryptography. Medium. [Online]. Available: https://medium.com/@rajeev.singh92/understanding-the-basics-of-cryptography-f377d1ab2e4b

16. Appendices

Appendix A: Code Listings

1. `main.py`: The main Python script containing the implementation of the file encryption system.
2. `vault.py`: Python module implementing the SecretVault class for managing encrypted files.
3. `db.py`: Python module implementing the Database class for SQLite database operations.

Appendix B: User Manual

The user manual provides detailed instructions on how to install, configure, and use the file encryption system. It includes step-by-step guides for encrypting and decrypting files, managing the vault, and troubleshooting common issues.

Appendix C: Sample Test Data

Sample test data includes a set of files used for testing the functionality of the file encryption system. It includes plaintext files for encryption, encrypted files for decryption, and various edge cases to test system robustness.

Appendix D: Glossary

The glossary contains definitions of key terms and concepts used throughout the report. It provides clarity and context for readers unfamiliar with technical terminology related to file encryption, cryptography, and software development.

Appendix E: Project Plan

The project plan outlines the timeline, milestones, and deliverables for the development of the file encryption system. It includes details on task allocation, resource requirements, and risk management strategies to ensure project success.

Appendix F: Stakeholder Communication

Stakeholder communication records include meeting minutes, progress reports, and feedback sessions with project stakeholders. These documents capture the collaborative decision-making process and ensure alignment with stakeholder expectations.

Appendix G: Code Documentation

The code documentation includes inline comments and documentation strings (docstrings) within the source code files. It provides additional context and explanation for code segments, enhancing readability and maintainability for future development efforts.