Software Requirements Specification

For

ZIP GUARD

18/10/2024

Prepared by

Specialization	SAP ID	Name
DevOps	500101788	Akshat Pandey
DevOps	500105699	Madhav Madan
DevOps	500106021	Dhruv Joshi
DevOps	500106891	Kavya Singh

UPES
UNIVERSITY OF TOMORROW

23/10/24

School Of Computer Science
UNIVERSITY OF PETROLEUM & ENERGY STUDIES,
DEHRADUN- 248007. Uttarakhand

22/10/24

Table of Contents

Revision History

Date	Change	Reason For Change	Mentor Signature

1. Introduction

1.1 Purpose of the Project

The purpose of the *Zip Guard* project is to develop an easy-to-use yet secure application that combines **file compression** and **encryption**. The application addresses two key concerns for today's digital users **file size optimization** and **data protection**. Through Huffman coding for file compression and AES/RSA for encryption, *Zip Guard* ensures that files can be reduced in size and safely encrypted for secure transmission and storage. By utilizing **auto-generated keys**, the system simplifies encryption processes for users while still maintaining a high level of security for sensitive files.

1.2 Target Beneficiary

The primary target audience includes:

- **Individual Users:** People who need to protect personal data, such as financial documents, personal records, and sensitive multimedia files.
- **Corporate Users:** Companies and organizations seeking to secure sensitive business data, intellectual property, and confidential client information.
- **Developers and Tech Enthusiasts:** Individuals or teams looking for a customizable solution to integrate file compression and encryption into larger applications.

1.3 Project Scope

'Zip Guard' offers a combination of file compression and encryption features, providing benefits such as reduced file size and enhanced security. The key deliverables of the project include:

- **File compression:** using Huffman coding, a lossless algorithm that reduces file size without losing data integrity.
- **Encryption:** using either **AES (symmetric encryption)** or **RSA (asymmetric encryption)**, depending on user needs.
- **Key generation and management:** with auto-generated public/private keys for RSA and symmetric keys for AES, allowing for secure file encryption without manual key input.

1.4 References

- [1] Oracle (2024) "Java™ Platform, Standard Edition Security Developer's Guide: StandardNames,"Availableat:https://docs.oracle.com/en/java/javase/22/docs/specs/security/standard-names.html .
- [2] Oracle (2024) "Java™ SE 8: Cryptography Roadmap," Available at: https://www.java.com/en/jre-jdk-cryptoroadmap.html .
- [3] Data Structures Project - File Compression Using Huffman Coding in JAVA (2023) "YouTube," Available at: https://www.youtube.com/watch?v=S0Wua5WxKZI .
- [4] IETF (2016) "PKCS #1: RSA Cryptography Specifications Version 2.2," RFC 8017. Available at: https://www.ietf.org/rfc/rfc8017.html .

2. Project Description

2.1 Reference Algorithm

- **Huffman Coding:** Huffman coding is a lossless compression algorithm that assigns variable-length codes to characters based on their frequency of occurrence. Characters with higher frequencies are assigned shorter codes, while less frequent characters get longer codes. This minimizes the overall size of the compressed data, allowing for efficient storage and transfer.
- **AES and RSA Encryption**

  AES (Advanced Encryption Standard): AES is a symmetric encryption algorithm, meaning it uses the same key for both encryption and decryption. It is highly efficient and is the standard for securing sensitive data.

  RSA (Rivest-Shamir-Adleman): RSA is an asymmetric encryption algorithm. It uses a public key for encryption and a private key for decryption, making it ideal for secure data exchange over untrusted channels.

2.2 Data / Data Structure

The Zip Guard project employs a tree structure for Huffman coding to efficiently compress files. The Huffman tree is constructed based on the frequency of characters in the input data, allowing for variable-length encoding that reduces file size. For encryption, RSA (Rivest-Shamir-Adleman) is utilized for asymmetric encryption, generating public and private key pairs for secure key exchange. In contrast, AES (Advanced Encryption Standard) is implemented for symmetric encryption, encrypting the compressed data blocks to ensure confidentiality. The overall data structure comprises compressed and encrypted data blocks, securely formatted for storage and transmission.

2.3 SWOT Analysis

- **Strengths**
  - Efficient file size reduction using lossless Huffman coding.
  - Secure encryption using AES or RSA to protect user data.
  - Auto-generated keys eliminate manual user intervention for encryption.
- **Weaknesses**
  - Processing large files more than size 128kb will be time consuming.
  - RSA encryption is computationally intensive for large data sets.
- **Opportunities**
  - Increasing demand for secure data transmission and storage in both personal and business sectors.
  - Potential for integration with cloud-based storage services.

- **Threats**
    - Emerging decryption techniques could threaten the long-term security of RSA and AES.
    - Existing competitors, like WinRAR and 7-Zip, offer built-in encryption features.

2.4 Project Features

The primary features of *Zip Guard* include:

- **File Compression:** Compresses files using Huffman coding to reduce their size.
- **File Encryption:** Users can choose between AES (symmetric) and RSA (asymmetric) encryption.
- **Automatic Key Generation:** Keys for AES and RSA encryption are auto-generated, simplifying usage for non-technical users.
- **File Decryption:** Allows users to decrypt previously encrypted files using the appropriate key.

2.5 User Classes and Characteristics

- **General Users:** Individuals who require easy encryption and compression solutions for personal data.
- **Corporate Users:** Organizations with sensitive business data that require encryption during storage or transmission.
- **Developers/Tech Enthusiasts:** Users who may require more customization options, such as key length and encryption strength.

2.6 Design and Implementation Constraints

- **Platform Dependency:** The project must be developed in Core Java, ensuring platform independence.
- **File Size Limitation:** The system should handle files as large as 128 kb efficiently.
- **Encryption Standards:** The application must use industry-standard encryption algorithms (AES and RSA).

2.7 Assumptions and Dependencies

- Users are assumed to have a basic understanding of file management (selecting, saving, and browsing files).
- The application relies on the Java Cryptography Extension (JCE) for handling encryption algorithms.
- The system is dependent on local storage for file management and encryption key storage.
- The file size is limited to 128kb.

3. System Requirements

3.1 User Interface Requirements

The user interface (UI) should be designed for ease of use, featuring:

- A file explorer for selecting files.
- Encryption options for users to choose between AES and RSA.
- Clear buttons for "Compress", "Encrypt", "Decrypt" actions.
- User must have the Java Development Kit (JDK) installed to run the application.

3.2 Software Interface

The software interfaces with:

- Java Cryptography Libraries for AES and RSA encryption.
- Huffman for file compression.
- GitHub integration for version control and collaboration.

3.3 Protocols

- **Encryption Protocols:**
  - **AES (Advanced Encryption Standard):** AES is utilized for symmetric encryption, providing fast and secure encryption of the compressed file data. This protocol is essential for ensuring that sensitive data remains confidential during storage and transmission.

  - **RSA (Rivest-Shamir-Adleman):** RSA is used as an asymmetric encryption protocol for secure key exchange. It enables the generation of a public-private key pair, allowing for secure transmission of the AES encryption key. This dual-layer encryption approach enhances overall data security.

- **Data Compression Protocol:** The Huffman coding algorithm serves as the data compression protocol. It compresses files based on character frequency, reducing storage space requirements and facilitating quicker uploads and downloads.

4. Functional Requirements

4.1 File Compression

The system will:

- Allow users to compress files using Huffman coding.
- Support various file types (e.g., text, image, video).

4.2 File Encryption

Users can:

- Choose between AES (symmetric) and RSA (asymmetric) encryption.
- Encrypt compressed files using selected algorithms.

4.3 Key Management

- RSA: The system generates public/private key pairs.
- AES: A symmetric key is generated automatically.

4.4 File Decryption

The user can:

- Decrypt files using the appropriate key (private for RSA, symmetric for AES).
- Receive warnings if an incorrect key is provided.

5. Non-functional Requirements

5.1 Performance Requirements

- Encryption speed should be optimized for files up to 128kb.
- Compression should maintain file integrity and provide a reduction in file size.

5.2 Security Requirements

- Use AES 256-bit encryption for strong data protection.
- Protect keys and encrypted files from unauthorized access.

5.3 Reliability and Availability

- The application must ensure that files are never lost during compression or encryption. Backup mechanisms may be added for larger, more critical files.

5.4 Portability

- The application must run on any Java-supported platform, including Windows, macOS, and Linux.

5.5 Scalability

- Future updates could include integration with cloud services for file storage, encryption, and decryption.

6. Implementation

The implementation of **Zip Guard** will follow a structured approach to meet the project's primary objectives. The key goals include efficient file compression using Huffman coding, offering flexible encryption options (RSA and AES), simplifying the key generation process, and ensuring secure file management.

6.1 Objective 1: Efficient File Compression using Huffman Coding

- **What:**
  - The goal is to implement **Huffman coding** for efficient file compression. Huffman coding reduces the size of files without losing any data by using shorter binary codes for frequently occurring characters.
- **How to Achieve:**
  - **Step 1**: Analyse the file and count the frequency of each character.
  - **Step 2**: Build a binary Huffman tree, where frequent characters are assigned shorter binary codes.
  - **Step 3**: Replace characters in the file with these binary codes to create the compressed file.
  - **Step 4**: Store both the compressed binary data and the Huffman tree for decompression later.
- **Completion Status:**
  - **Achieved:** Huffman coding has been successfully implemented.

6.2 Objective 2: Provide RSA Encryption Option

- **What:**

  Provide users with the option to use **RSA (asymmetric encryption)** for securing their compressed files. RSA ensures that data can only be decrypted by the intended recipient.
- **How to Achieve:**
  - **Step 1**: Automatically generate a public and private key pair for each user.
  - **Step 2**: Use the public key to encrypt the compressed file.
  - **Step 3**: The user or recipient will use the private key to decrypt the file securely.
- **Completion Status:**
  - **Achieved:** RSA encryption has been successfully implemented. Users can now encrypt and decrypt files using RSA.

6.3 Objective 3: Automate the Key Generation Process

- **What:**
  - Automatically generate keys (RSA public/private keys and AES secret keys) to simplify the encryption process for users without technical expertise.
- **How to Achieve:**
  - **Step 1**: For RSA, automatically generate the public and private key pair when RSA is chosen.
  - **Step 2**: For AES, generate a random secret key when AES encryption is selected.
  - **Step 3**: Securely store the generated keys within the system.

- **Completion Status:**
  - **Pending:** Automated key generation is still under development. It will be integrated with both RSA and AES encryption options.

6.4 Objective 4: Integrate Encryption, Decryption, Compression, and Decompression into a Single, Cohesive Package

- **What:**
  - The aim is to create a unified system that seamlessly integrates file compression (using Huffman coding) with encryption (using AES or RSA) and provides corresponding decryption and decompression functionalities. This ensures that users can easily compress, encrypt, decrypt, and decompress their files within one tool.

- **How to Achieve:**
  - **Compression (Huffman Coding):** Compress the file to reduce its size by encoding frequent characters with shorter binary codes.
  - **Encryption (AES/RSA):** After compression, the user selects either AES or RSA encryption.
    - **AES:** Encrypt the compressed file using the AES symmetric encryption algorithm.
    - **RSA:** Encrypt the compressed file using RSA's asymmetric encryption.
  - **Decryption:** The reverse process, where encrypted files are decrypted using the appropriate key (AES key or RSA private key).
  - **Decompression:** Once decryption is complete, the file is decompressed using the saved Huffman tree, restoring it to its original form.

- **Completion Status:**
  - **Partially Achieved:** Compression and Decompression (Huffman) and RSA encryption have been fully implemented and integrated. AES encryption functionalities are under development, with future iterations aiming to complete and integrate all functionalities into one streamlined package.

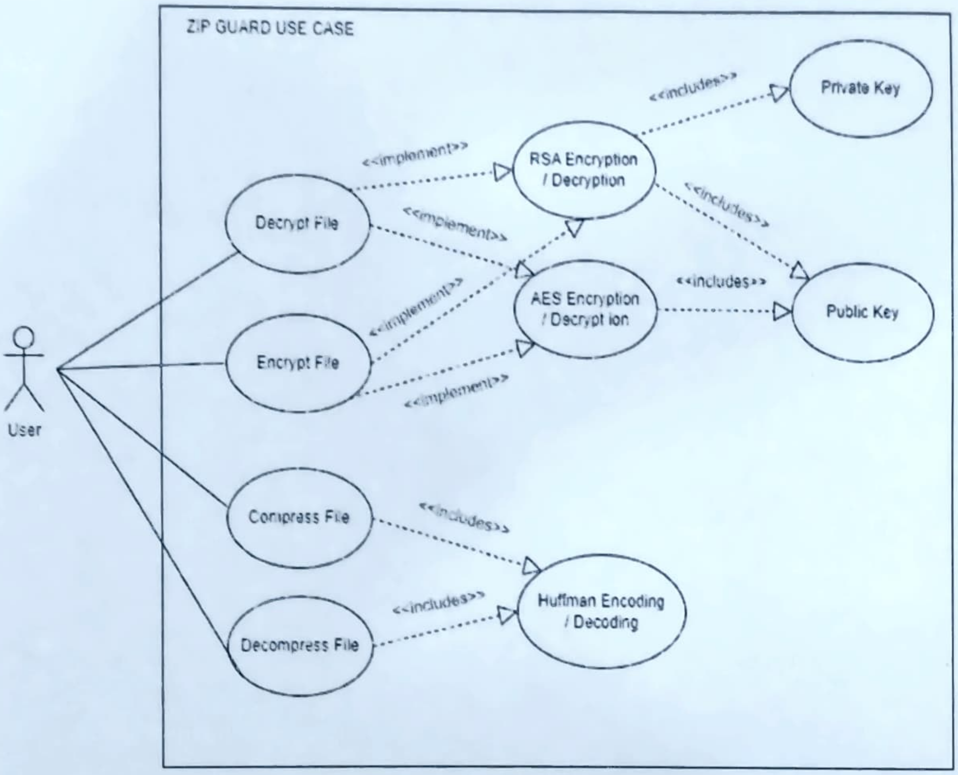
Appendix A: Glossary

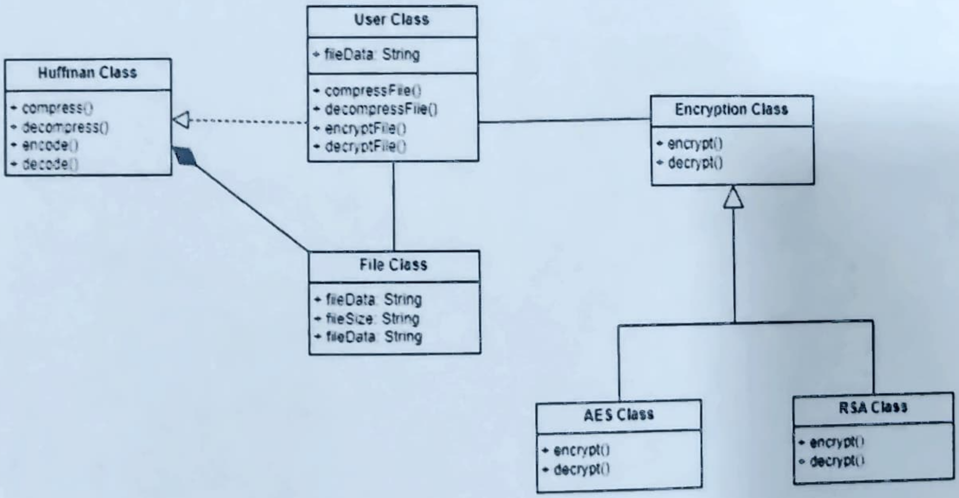
- **AES (Advanced Encryption Standard):** A symmetric encryption algorithm used worldwide for securing sensitive data.
- **RSA (Rivest-Shamir-Adleman):** An asymmetric cryptographic algorithm widely used for secure data transmission.
- **Huffman Coding:** A lossless data compression algorithm that is commonly used to reduce file sizes.

- **Encryption:** The process of converting plaintext into a coded form to prevent unauthorized access.
- **Decryption:** The process of converting encrypted data back to its original form.
- **Symmetric Encryption:** An encryption method in which the same key is used for both encryption and decryption.
- **Asymmetric Encryption:** An encryption method that uses a public key for encryption and a private key for decryption.

Appendix B: Analysis Model

- Includes detailed diagrams such as, Use Case and Class diagram.


ZIP GUARD USE CASE
User
Decrypt File
Encrypt File
Compress File
Decompress File
RSA Encryption / Decryption
AES Encryption / Decrypt ion
Huffman Encoding / Decoding
Private Key
Public Key
<<implement>>
<<implement>>
<<implement>>
<<implement>>
<<includes>>
<<includes>>
<<includes>>
<<includes>>
<<includes>>


User Class
+ fileData: String
+ compressFile()
+ decompressFile()
+ encryptFile()
+ decryptFile()
Huffman Class
+ compress()
+ decompress()
+ encode()
+ decode()
Encryption Class
+ encrypt()
+ decrypt()
File Class
+ fileData: String
+ fileSize: String
+ fileData: String
AES Class
+ encrypt()
+ decrypt()
RSA Class
+ encrypt()
+ decrypt()