Table of Figure

1. Introduction

In the digital era, securing sensitive information has become more important than ever. The File Encryptor in C++ project is designed to provide a reliable way to protect confidential data by converting readable files into an unreadable format using encryption algorithms. This ensures that even if unauthorized users gain access to the files, they cannot understand or misuse the information without the correct decryption key or password.

The project is not only a demonstration of basic cryptography but also serves as a practical example of file handling, UI design using WinBGIm, and implementing basic security protocols.

1.1 Overview

The **File Encryptor** project is a desktop-based application developed in C++ that allows users to securely encrypt and decrypt files, helping protect sensitive data from unauthorized access. In an era of increasing concern over data privacy and cybersecurity, this tool provides a practical and educational solution for safeguarding personal or confidential information using a basic encryption algorithm—the Caesar Cipher.

Users can select a file through a graphical interface, choose an operation (encrypt or decrypt), and the program will transform the file content into an unreadable format (cipher text), which can only be restored using the correct decryption method. The system includes key features such as file validation, visual feedback, error handling, and operation history logging to ensure a smooth and user-friendly experience.
Designed for simplicity and clarity, this project demonstrates the core concepts of file encryption and offers a strong foundation for students and beginners to understand and experiment with cryptographic principles.

1.2 Problem statement

In the current digital landscape, data is frequently shared and stored across various platforms, often without adequate protection. Many users, especially individuals and small organizations, rely on basic file storage methods that do not include any form of encryption. As a result, sensitive files are vulnerable to unauthorized access, data breaches, and cyber-attacks. Existing commercial encryption tools may be costly, complex, or require internet connectivity and advanced technical knowledge, making them less accessible to general users or students.

Moreover, some open-source or built-in file encryption solutions offer limited customization, lack transparency in their encryption processes, or provide minimal control over key management. This creates a significant gap for users who need a lightweight, efficient, and easy-to-use encryption tool for protecting personal or confidential data. There is a clear need for a standalone, platform-independent file encryptor that prioritizes security, simplicity, and control addressing the shortcomings of current systems while being accessible for educational and practical use.

1.3 Features

File Encryption & Decryption
Encrypts and decrypts files using the Caesar Cipher with a fixed shift value.

Graphical User Interface(GUI)
User-friendly interface built with graphics.h, allowing navigation through menus with mouse clicks.

File Selection Dialog
Users can browse and select files using a standard Windows file dialog box.

Loading Animation
Displays a loading animation during encryption/decryption to indicate progress.

Sound Feedback
Plays success or error sounds upon completion of operations for better user interaction.

History Logging
Logs every encryption or decryption action along with a timestamp in a history.txt file.

Error Handling
Displays messages for invalid files, missing input, or operation failure.

Visual Confirmation
Message boxes and on-screen prompts confirm each action and outcome.

Supports Text and Binary Files
Can handle different file types, ensuring flexibility in usage.

Lightweight and Standalone
Does not rely on external libraries (except graphics.h) and runs as a single executable.

1.4 Objective

Confidentiality:
Ensure that the contents of a file cannot be understood by unauthorized users.
Encryption transforms readable data (plaintext) into unreadable data (ciphertext).

Data Integrity:
Prevent undetected modification of the file during storage or transmission.
While encryption alone doesn't guarantee integrity,

Access Control:
Only authorized users with the correct decryption key can access the file’s original content.

Secure Storage:
Protect sensitive data stored on disk, such as user credentials, personal information, or business documents.

Secure Transmission:
Ensure the file remains protected during network transmission, preventing eavesdropping.

1.5 Scope and Limitation

1.5.1 Scope

- Provide a **simple yet functional** tool to encrypt and decrypt files.
- Help users **protect files** from unauthorized access.
- Implement the **Caesar Cipher algorithm** for:
    - Text files
    - Binary files
- Allow users to:
    - **Select a file**
    - Choose an operation: **Encryption** or **Decryption**
    - Generate a **new output file**
- Built using **C++** with the **WinBGIm graphics library**.
- GUI features include:
    - **File selection dialog**
    - **Loading animations**
    - **Sound effects**
    - **History logging**
- Target audience:
    - **Students and beginners** learning file handling and encryption
- Educational value:
    - Demonstrates real-world use of **cryptographic operations**
- Current limitations:
    - Only uses **Caesar Cipher**
    - Works only on **Windows**
- Future extension possibilities:
    - **Stronger encryption algorithms**
    - **Password protection**
    - **Cross-platform support**

1.5.2 Limitation

- The program does not support network or cloud-based file encryption.
- It uses a fixed encryption algorithm and does not allow algorithm switching.
- Does not store or manage keys securely key/password must be remembered by the user.
- It may not work properly with very large files due to memory usage.
- Does not provide protection against advanced attacks like side-channel or brute-force unless combined with strong passwords.

2. Methodology

The **File Encryptor** in C++ was developed using a modular and iterative approach. The project followed Agile principles, breaking the work into small sprints focusing on core features like encryption logic, file handling, and the graphical user interface. Early prototypes were built to test Caesar Cipher functionality before integrating advanced features such as file selection dialogs, sound effects, and history logging. Each module was tested individually, and peer feedback was used to refine the design. This method ensured a clear development process and a reliable, user-friendly final product.

2.1 Development Approach: Agile + Modular Prototyping

The development of the File Encryptor in C++ project followed a structured, modular approach incorporating Agile principles and iterative prototyping. The methodology was tailored to the nature of system utility software development, emphasizing quick testing, continuous feedback, and incremental functionality. The project was developed in sprints, each focusing on a specific functional component such as encryption logic, user interface, or file handling.

Key Phases in Development:

I. **Requirement Planning:**
- Core functionalities such as Caesar Cipher encryption/decryption, file I/O operations, and UI interaction were identified and scoped.
- Features were categorized as:
- Core Features: File selection, Caesar Cipher logic, encryption/decryption execution, file output, success/error handling.
- Stretch Features: Graphical user interface using WinBGIm, history logging, sound feedback, file validation.

II. **Rapid Prototyping:**
- Early prototypes focused on verifying Caesar Cipher logic using console-based input/output.
- Initial file reading/writing mechanisms were implemented and tested on small .txt files.
- Prototypes helped validate the shift-key logic and ensured that special characters remained unaffected.

III. **Incremental Feature Development:**
- Components were developed incrementally in logically isolated modules:
- Encryption Module: Handles Caesar Cipher transformations with support for both uppercase and lowercase alphabets.
- File Handler: Manages reading from and writing to user-specified files in binary mode.
- UI Layer: Developed using the WinBGIm graphics library to allow user navigation, file selection, and visual feedback.

- Logger Module: Appends each successful operation (with timestamp) to a history.txt file.
- C++ namespaces and classes were used to ensure clean code separation and maintainability.

IV. **Testing and Feedback:**

- Each module was tested independently for:
- File access reliability.
- Correctness of encryption/decryption output.
- Response to edge cases (empty files, non-text characters).
- Peer testing was conducted to ensure the GUI was user-friendly and logically structured.
- Feedback led to enhancements in error messages and improvements in visual animations.

V. **Integration and Optimization:**

- All independent modules were combined into a unified application with a main menu and user interaction loop.
- Load animations, click-detection logic, and file dialogs were integrated seamlessly.
- Audio cues and loading indicators were added to improve user experience and accessibility.

VI. **Final Polishing and Maintenance:**

- Final version was cleaned of bugs, optimized for basic performance, and packaged for demonstration.
- Codebase was documented with comments and structured headers for maintainability.
- Future improvements may include:
- Password-based encryption.
- Support for multiple algorithms.
- Cross-platform GUI integration (beyond graphics.h).

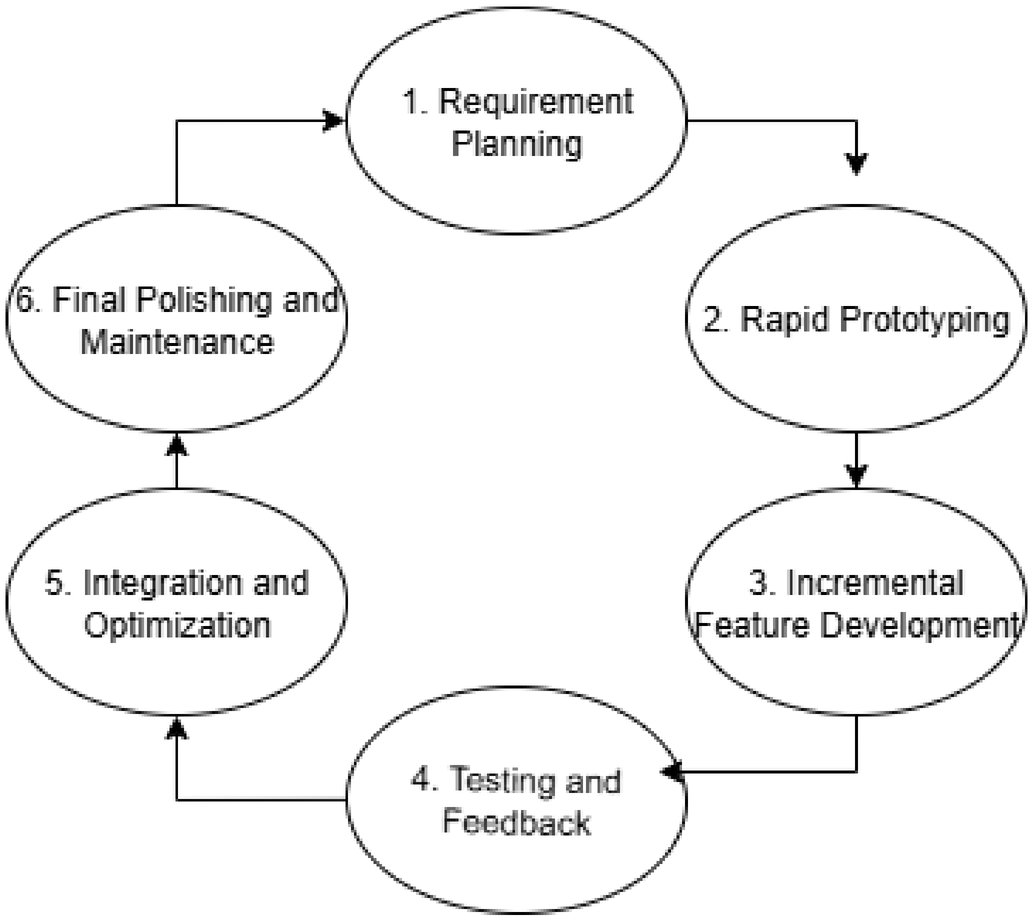
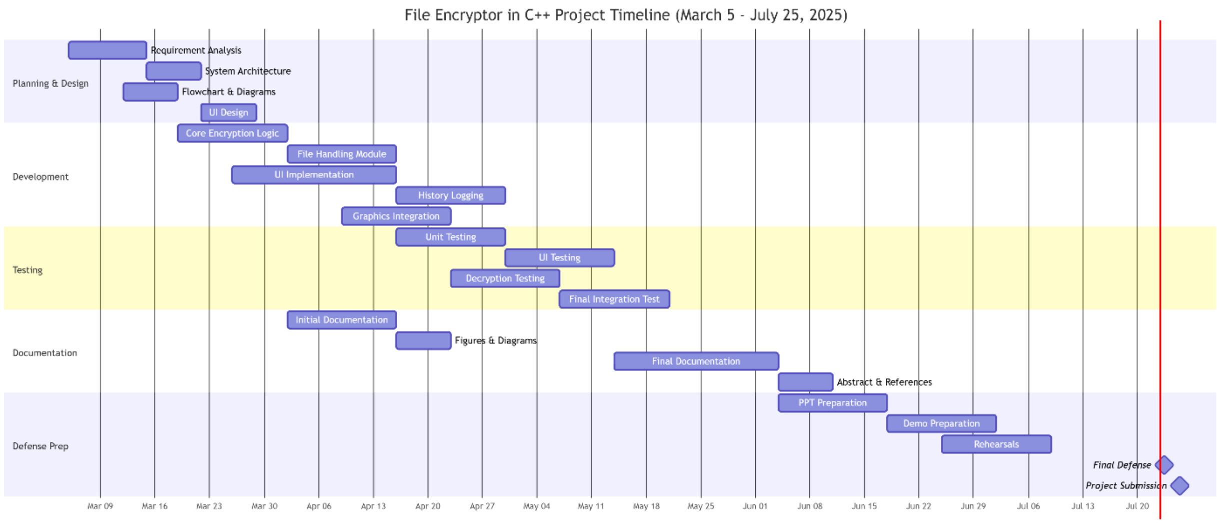

1. Requirement Planning
2. Rapid Prototyping
3. Incremental Feature Development
4. Testing and Feedback
5. Integration and Optimization
6. Final Polishing and Maintenance

Fig 1: agile modular prototyping

2.2 Gantt chart



2.3 Technologies and Tools Used

Technologies and Tools used for the File Encryptor Project

SN	TOOLS	PURPOSE
1	C++	Core programming language
2	VS Code, DEV C	Write, debug, and run code
3	Compiler (GCC, MSVC)	Turn code into executable
4	Standard Libraries	File I/O, encryption logic
5	Operating system	Any major OS (Windows/Linux/macOS)

Fig 2: Table of file encryptor in c++

3. System Design

3.1 System Architecture

The system follows a modular architecture:

- **UI Layer**: Built with WinBGIm, provides main menu, animations, file dialog.
- **Core Logic**: Handles Caesar Cipher encryption and decryption.
- **File Handling**: Reads input and writes output securely.
- **Logging Module**: Records actions in history.txt.
- **Feedback**: Provides user interaction via sound and visual effects.

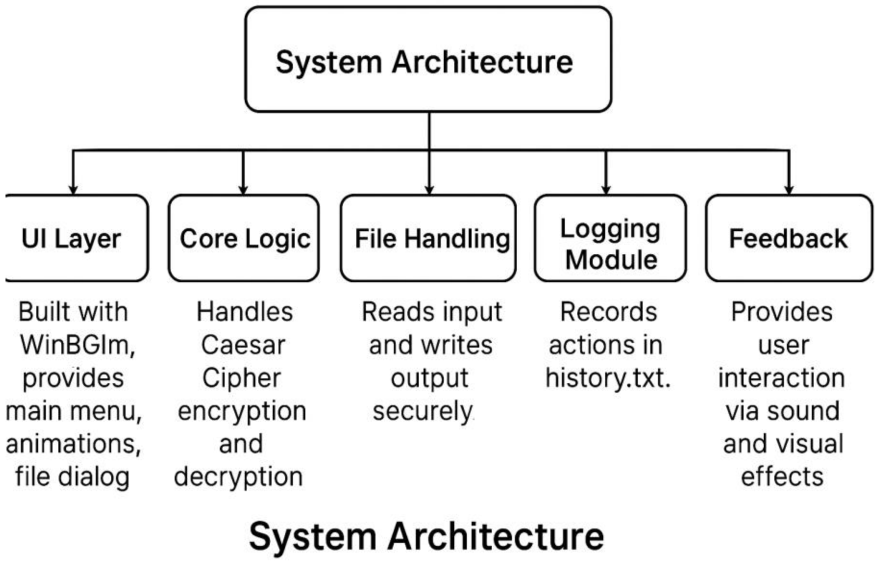

System Architecture
UI Layer
Built with WinBGIm, provides main menu, animations, file dialog
Core Logic
Handles Caesar Cipher encryption and decryption
File Handling
Reads input and writes output securely
Logging Module
Records actions in history.txt.
Feedback
Provides user interaction via sound and visual effects
System Architecture

Fig: 3: system architecture 1

3.2 Flowchart

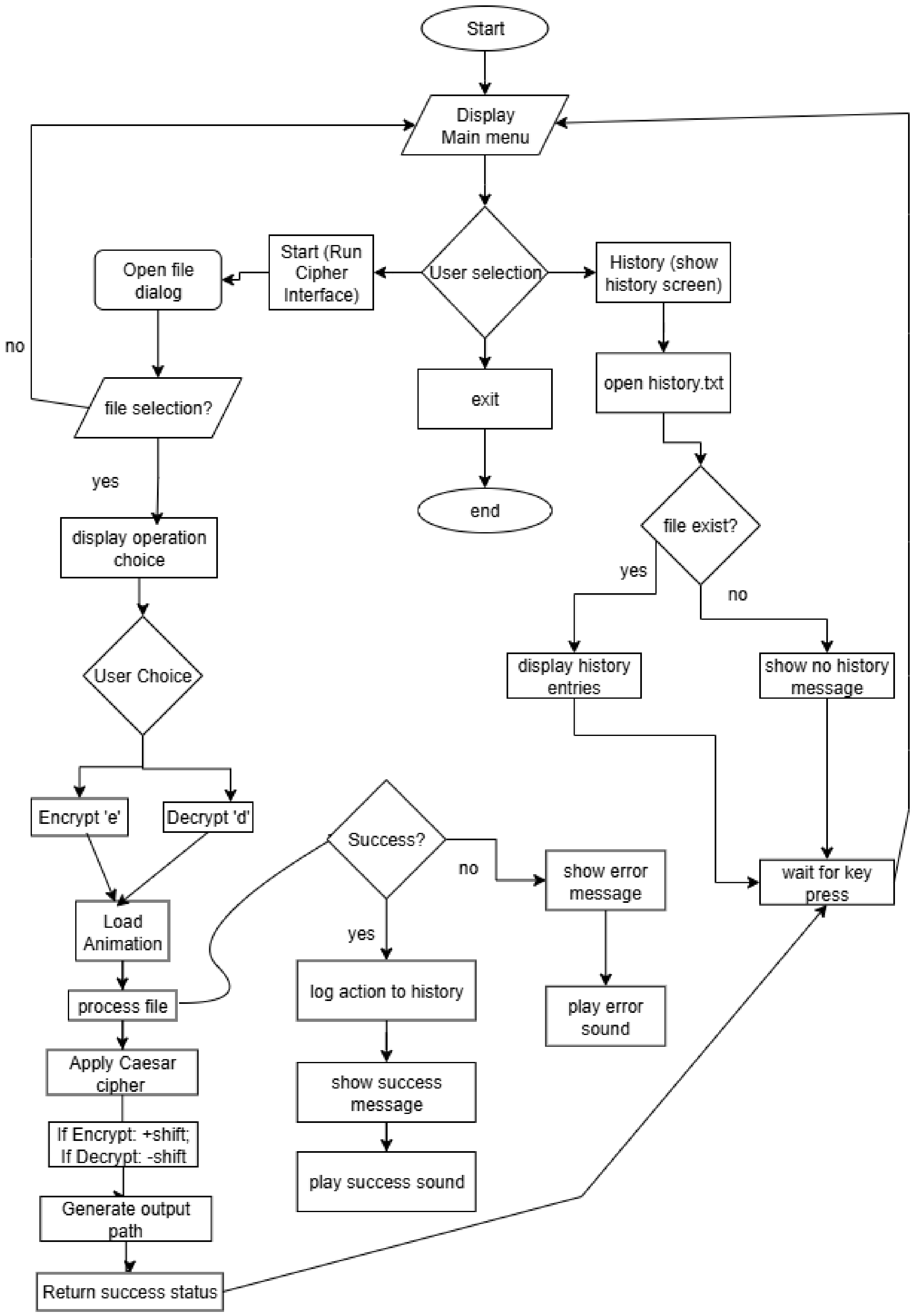


Fig 4: Flowchart of file encryptor in c

4. Object Oriented

4.1 Class Diagram

The Caesar Cipher is one of the oldest and simplest encryption algorithms. It works by shifting each letter in the plaintext by a fixed number of positions in the alphabet. For example, with a shift of 3, the letter 'A' becomes 'D', 'B' becomes 'E', and so on. If the shift goes beyond 'Z', it wraps around to the beginning of the alphabet, so 'Z' would become 'C'. To decrypt the message, the letters are shifted back in the opposite direction by the same number. Characters that are not letters, such as numbers or punctuation, are typically left unchanged. The Caesar Cipher is a form of substitution cipher and is very easy to break using brute-force or frequency analysis techniques.

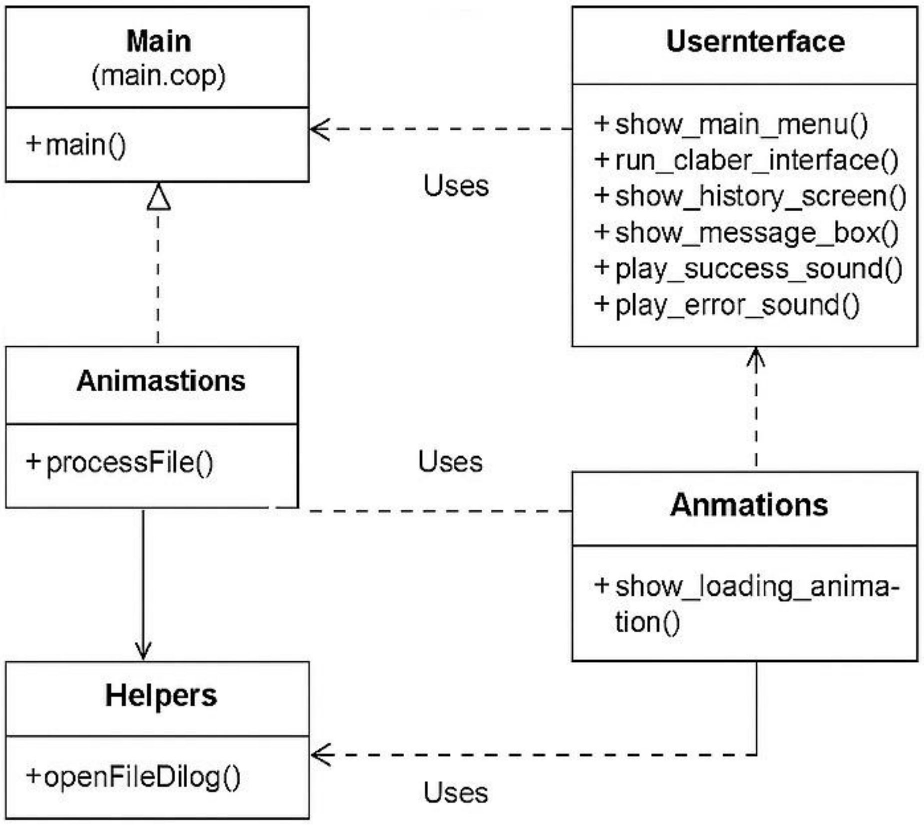


Fig 5: class diagram of file encryptor

4.2 Use Case Diagram

Encryption Tool:

A C++ desktop application that allows users to encrypt and decrypt files with additional user-interface feedback features.

Primary Actor

- **User** – Interacts with the Encryption Tool through the graphical menu interface.

Use Cases and Descriptions

Use Case	**Description**
Select File	The user chooses a file from their system to be processed.
Encrypt File	The system applies a Caesar cipher to the selected file to produce an encrypted output.
Decrypt File	The system reverses the encryption on a selected file to restore the original content.
View History	The user can view a log of previously performed encryption and decryption actions.
Play Sound Feedback	The system plays success or error sounds to give immediate feedback after an operation.
See Loading Animation	The system shows a simple loading animation while processing to indicate progress.

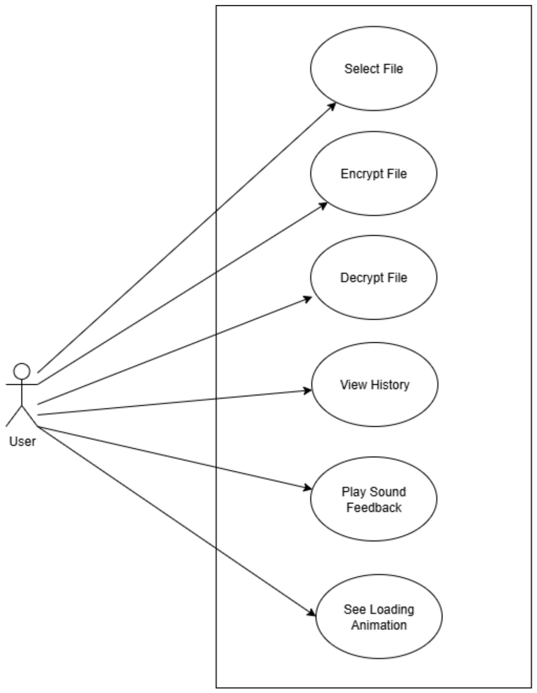


Fig 6: use case diagram of file encryptor

4.3 Activity Diagram

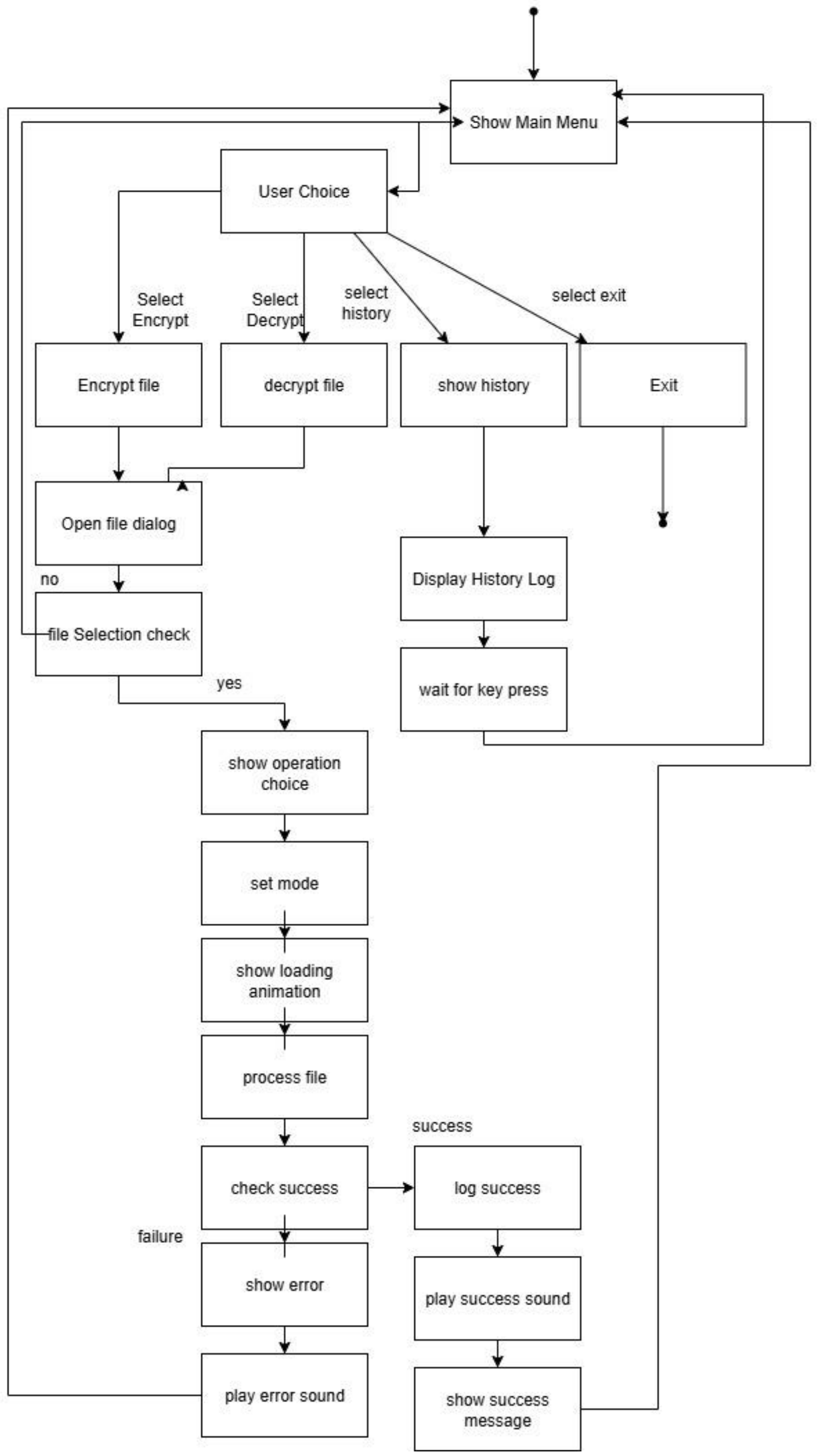


Fig 7: Activity diagram of file encryptor

4.4 Sequence Diagram

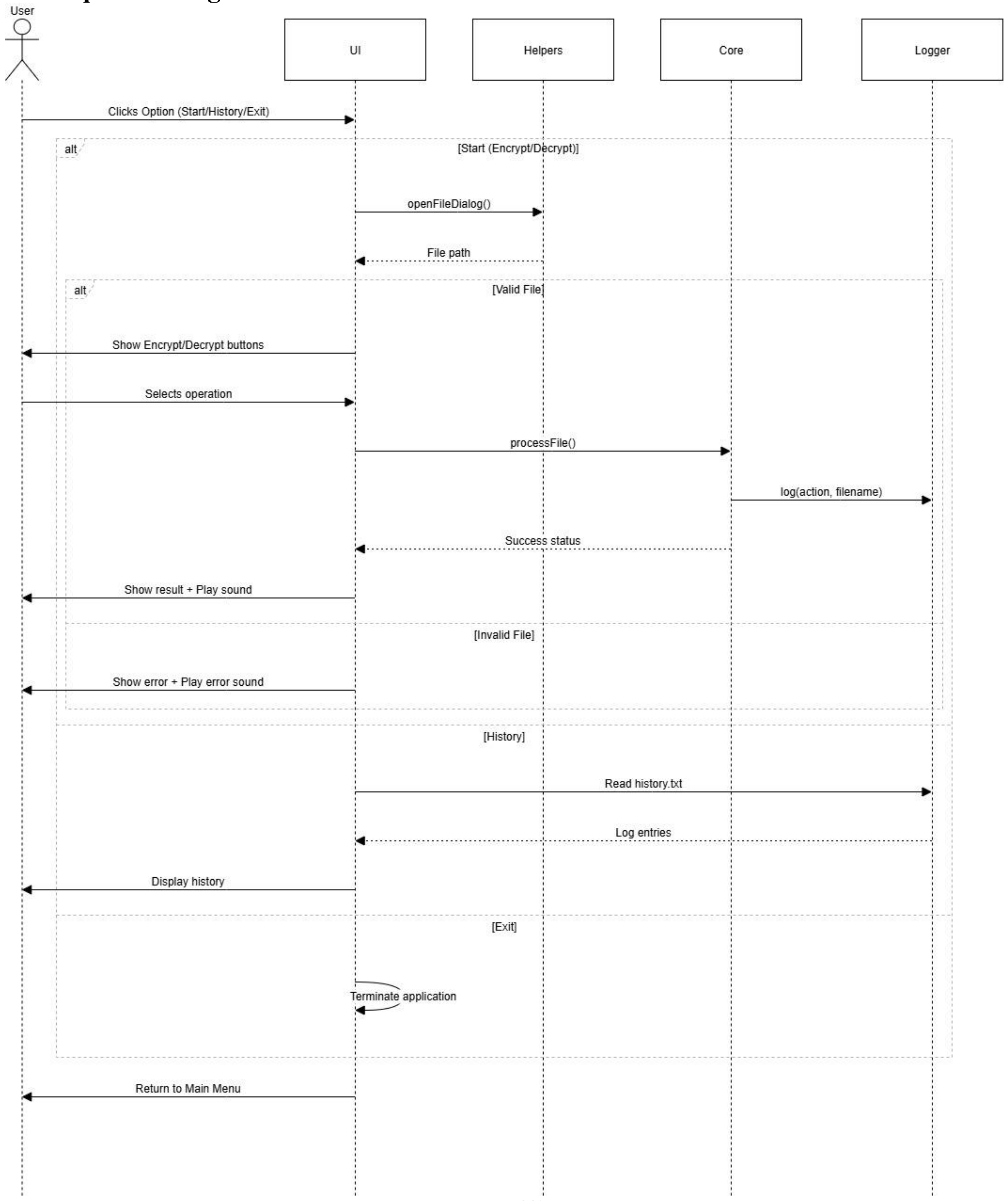


Fig 8: Sequence Diagram of file encryptor

5. System Development and Implementation

Programming Platform

The **File Encryptor** project, implemented using the Caesar Cipher algorithm in C++, was developed on a simple yet efficient programming setup.

C++ was chosen for its performance, portability, and extensive standard library support. The development environment included **Visual Studio Code** and **Dev-C++ IDE**, both of which provide a clean interface, syntax highlighting, and debugging features that make development easier and more organized.

For compilation, standard compilers such as **MSVC** (Microsoft Visual C++) and **GCC/MinGW** were used. These compilers ensure that the code runs efficiently on Windows systems. The project was designed and tested primarily on a **Windows operating system**, making use of its straightforward file handling and graphics support.

The tool works with basic text files (.txt) as input and output for encryption and decryption, which makes testing and usage simple.

No third-party libraries were required—only standard C++ libraries—keeping the project lightweight and beginner-friendly.

Operating Environment

The **File Encryptor** is lightweight and can run on any modern computer with a basic setup. It is compatible with popular operating systems such as **Windows**, **Linux**, or **macOS**, provided that a C++ compiler (e.g., **GCC**, **MinGW**, or **MSVC**) is installed.

Development and usage do not demand high-end hardware. A system with at least:

- **2 GB RAM**,
- a **dual-core processor**, and
- sufficient disk space to store input and output files

is adequate to run the application smoothly.
The tool can be built and executed using common code editors or IDEs like **Visual Studio Code**, **Dev-C++**, **Code::Blocks**, or even **Notepad++** for quick edits.

The only requirement is that the system must have permission to read from and write to the local disk where the input and output files are stored.

All testing was performed using standard text files to ensure that both encryption and decryption functions work correctly. Because no special libraries or frameworks are required, the project is easy to set up, portable, and ideal for learning or demonstration purposes.

6. Assignment of Roles and Responsibilities

Member Name	**Role & Responsibilities**
Saraswati Rokaya	**UI & User Interaction:** Designed the user interface (menus, buttons, messages), implemented graphical feedback (loading animation, message boxes), and integrated sound feedback for user actions.
Salim Shrestha	**Core Logic & Integration:** Implemented the main encryption/decryption logic using the Caesar Cipher, managed file input/output operations, and integrated all modules into a working system.
Aayush Kumar Mallik	**Testing & Documentation:** Performed testing with different files to ensure correct functionality, debugged issues, maintained history log features, and prepared project documentation including diagrams and proposal sections.

Fig 9: Table of Role & Responsibilities

7. Testing and Debugging

Testing and debugging ensured the correct functionality of file encryption, decryption, and user interface components. Various test cases were executed to handle valid inputs, edge cases, and unexpected user actions without crashing the application.

7.1 Testing Approach

The **File Encryptor** project was thoroughly tested to ensure that all core features work as expected, with testing carried out using both valid and invalid input files to cover different scenarios.

The main objectives of testing were:

- To verify that encryption and decryption produce correct results.
- To confirm that the user interface responds properly to all interactions.
- To ensure that the history log, sound feedback, and loading animation work smoothly without crashing.

7.2 Test Cases:

Test Case	**Description**	**Expected Result**	**Outcome**
Encrypt valid file	Select a .txt file and encrypt it	Encrypted file (encrypted_filename.txt) is created	Pass
Decrypt valid file	Select an encrypted file and decrypt it	Original content restored in decrypted_filename.txt	Pass
View history	Open history log after several operations	All actions are listed with timestamps	Pass
Cancel file selection	Close file dialog without selecting a file	Show message and return to main menu	Pass
Missing file	Select a non-existing file	Error message displayed, no crash	Pass

Fig 10: test cases

7.4 Debugging Process

During development, debugging was performed iteratively:

1. **Compile-time Errors:**
   Fixed syntax errors and type mismatches during compilation using IDE error messages and compiler feedback.
2. **Runtime Testing:**
   - Ran the program step by step to identify logic issues.
   - Checked edge cases like empty files or special characters.
   - Verified that file handles were properly closed after operations.
3. **UI and Graphics Debugging:**
   - Adjusted button coordinates and click regions.
   - Tested sound file paths and confirmed animations did not overlap text.
4. **Logging and Verification:**
   - Cross-checked history.txt to verify accurate encryption/decryption records.

8. Conclusion

In this project, we successfully developed a file encryption system using the Caesar Cipher algorithm in C++. The Caesar Cipher, one of the simplest forms of encryption, involves shifting each letter in the text by a fixed number of positions in the alphabet. While the method itself is basic and not suitable for advanced security applications, it served as an effective way to explore the core principles of encryption.

Through the process of reading file content, applying the Caesar Cipher transformation, and writing the encrypted or decrypted output to a new file, we gained practical experience in file handling, user interaction, and implementing logical operations in C++. Additionally, we integrated a graphical user interface using the WinBGIm library, along with sound and visual feedback, to enhance user experience.

Overall, this project provided us with a solid foundation in understanding how cryptographic systems function and how security features can be integrated into real-world applications. It has also prepared us to explore more complex encryption techniques and data protection strategies in future projects.

9. References

- TutorialsPoint. (n.d.). Caesar Cipher Algorithm in C++. Retrieved March 13, 2025, from https://www.tutorialspoint.com/caesar-cipher-in-cplusplus
- WinBGIm Documentation. (n.d.). WinBGIm Graphics Library. Retrieved March 14, 2025, from http://winbgim.codecutter.org/
- GeeksforGeeks. (n.d.). File Handling in C++. Retrieved March 14, 2025, from https://www.geeksforgeeks.org/file-handling-in-cpp/