

PROJECT TITLE

Image Encryption-Decryption Mobile Application

By

**Hadi Mcheimech**

Submitted to the School of Arts & Science of the

Lebanese International University Nabatieh, Lebanon

In part of fulfillment of the requirements for the degree of

**BACHERLOR OF SCIENCE IN**

**COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**

**Supervised by: Dr. Reda Shbib**

Spring 2024-2025

# **Dedication:**

First of all, we will dedicate this project to our almighty God, who gave us strength and knowledge for our everyday life.

To our beloved parents and friends, for always supporting us in our studying and work, and for helping us find and realize our potential.

Last to our respected Doctor Reda Shbib for leading us step by step to reach our aim and to overcome every single obstacle we faced during the time we worked on our project and even before that during our three years of studying.

# **Acknowledgment:**

The development of the **"PixelLock"** project has been a challenging yet rewarding experience. It required significant effort, dedication, and collaboration. However, its successful completion would not have been possible without the invaluable support and contributions of many individuals and organizations.

We would like to express our deepest gratitude to our professor, **Dr. Rida Shbib**, for his expert guidance, continuous supervision, and for providing the critical insights and information that shaped our project. His dedication to our academic and personal growth has been truly inspiring.

We are also sincerely thankful to our parents and friends for their encouragement, emotional support, and belief in us throughout this journey. Their presence made the process smoother and more meaningful, and we are truly grateful for their role in our success.

# **Abstract:**

Our project is a mobile application **(PixelLock)** designed for **secure image encryption and decryption**, ensuring user privacy and data protection through robust cryptographic techniques and intelligent access control. The app provides a simple yet powerful way for users to encrypt images from their gallery or camera, secure them with a custom password, and store them safely on their device.

At the core of the system is **AES encryption in GCM mode**, with **keys derived securely using PBKDF2** and a **randomly generated salt and IV** (Initialization Vector). Encrypted files are marked with a custom header and saved with a **.enc** **extension**. Users can assign names to encrypted files, and the app supports real-time encryption progress indication.

To enhance usability and security, the app includes a **biometric lock screen** that leverages device-supported authentication (e.g., fingerprint scan) and **secure password validation using SHA-256 hashing** and persistent storage via Flutter Secure Storage. Multiple failed attempts trigger timed lockouts, with increasing durations and an optional **self-destruct mode** after too many incorrect password attempts, offering **maximum protection in sensitive environments**.

Additional features include:

* A responsive UI for password input and validation.
* Secure local history tracking of encrypted images (with hashed passwords).
* Cross-platform file picker support for folder selection and image access.
* Intuitive decryption flow with validation of the custom file format.

This application blends **cryptographic strength, modern biometric integration, and user-centric design** to deliver a reliable and user-friendly solution for **protecting personal and sensitive visual data on mobile devices.**

**Contents**

[Dedication: II](#_Toc199343695)

[Acknowledgment: II](#_Toc199343696)

[Abstract: III](#_Toc199343697)

[Table of Figures V](#_Toc199343698)

[Table of Tables VI](#_Toc199343699)

[Chapter:1 VII](#_Toc199343700)

[Introduction VII](#_Toc199343701)

[Objective VII](#_Toc199343702)

[Technology Constraints VII](#_Toc199343703)

[Problem Statement VIII](#_Toc199343704)

[Proposed Solution VIII](#_Toc199343705)

[Chapter: 2 IX](#_Toc199343706)

[Functional and Non-Functional Requirements: IX](#_Toc199343707)

[1)Functional Requirements IX](#_Toc199343708)

[2) Non-Functional Requirements X](#_Toc199343709)

[Implementation XI](#_Toc199343710)

[**1)** Application Initialization Logic XI](#_Toc199343711)

[**2)** Password Setup Workflow XI](#_Toc199343712)

[**3)** Lock Screen Authentication Logic XIII](#_Toc199343713)

[**4)** Post-Authentication Navigation XV](#_Toc199343714)

[**5**) Home Page Functionality **(MyHomePage)** XV](#_Toc199343715)

[**6)** Password Input Page XIX](#_Toc199343716)

[**7**) Image Controller Logic XXII](#_Toc199343717)

[**8**) Image Encryption Logic (ImageEncryptor) XXV](#_Toc199343718)

[**9**) Image Decryption Logic (ImageDecryptor) XXVII](#_Toc199343719)

[**10)** Encryption History Viewer (HistoryPage) XXX](#_Toc199343720)

[**11**) Permissions and Dependencies XXXIII](#_Toc199343721)

[Chapter: 3 XXXV](#_Toc199343722)

[Conclusion XXXV](#_Toc199343723)

[Future Work XXXV](#_Toc199343724)

# **Table of Figures**

[Figure1 : Create new Password Screen 1 XI](#_Toc199343011)

[Figure2: Login Page 1 XIII](#_Toc199343090)

[Figure3: Home Page Screen 1 XVII](#_Toc199343128)

[Figure4: Input Password Page 1 XX](#_Toc199343140)

[Figure5: Choose Folder 1 XXIII](#_Toc199343150)

[Figure6: Encrypting Process 1 XXV](#_Toc199343161)

[Figure7: Decrypting Password 1 XXVIII](#_Toc199343171)

[Figure8: History Page 1 XXXI](#_Toc199343186)

# **Table of Tables**

[Tabel1: Table of Dependencies 1 XXXIII](#_Toc199343518)

[Tabel2: Tabel of Permissions 1 XXXIV](#_Toc199343523)

## 

# Chapter:1

## Introduction

In the modern digital world, smartphones store vast amounts of personal data, making sensitive images vulnerable to unauthorized access. Many users lack proper tools to secure their private media, leading to potential data breaches. This project presents a secure and user-friendly mobile app that allows image encryption and decryption using robust AES encryption and biometric authentication. It includes password protection, fingerprint login, real-time progress tracking, and a self-destruct feature after repeated failed attempts offering a safe and private way to protect personal images.

## **Objective**

This project aims to develop a secure mobile app for encrypting and decrypting images. It uses password and biometric authentication, with AES-GCM encryption and PBKDF2 for strong protection. The app includes features like lockout timers, self-destruct after failed attempts, and a simple, user-friendly interface to ensure both security and ease of use.

Technology Constraints  
**Software:** Android Studio, Visual Studio Code.  
**Languages:** Dart, Java (Flutter).  
**Libraries & Packages:** encrypt, pointycastle, local\_auth, file\_picker, image\_picker, shared\_preferences, flutter\_secure\_storage.  
**Hardware:** Android smartphone, Laptop.  
**Platform:** Android (Mobile).

## **Problem Statement**

As smartphones store more sensitive content, securing personal images is more critical than ever, and that’s why:

* Personal images on phones are prime targets for hackers and spyware.
* Weak passwords and unencrypted storage make theft easy.
* Cloud sync and app flaws increase exposure to cyber threats.
* Most apps lack strong encryption and proper access control.
* A secure, user-friendly solution is urgently needed.

## **Proposed Solution**

We propose a secure mobile app for offline image encryption and decryption, designed for privacy-conscious users. The app uses robust cryptographic standards and a seamless user interface for non-technical users.

Key features included:

* **AES-GCM encryption** with PBKDF2 key derivation for advanced data protection.
* **Biometric authentication** for fast and secure access.
* **Custom lock screen** with hashed password and lockout functionality.
* **Failed login handling**, with increasing delays and a final self-destruct option.
* **Secure local history** of encrypted files for easy reference.

This solution combines strong security with ease of use, ensuring safe image storage and user peace of mind.

# Chapter: 2

## Functional and Non-Functional Requirements:

### 1)Functional Requirements

1. **User Authentication:**

* Login using password and/or biometric (fingerprint/face) authentication.

1. **Image Selection:**

* Ability to select images from gallery or capture from the camera.

1. **Image Encryption:**

* Encrypt selected images using AES-GCM with password-based key derivation (PBKDF2).
* Allow users to assign custom file names and passwords for encryption.

1. **Image Decryption:**

* Decrypt encrypted images upon correct password.
* Validate file format and encryption metadata.

1. **Secure Storage:**

* Store encrypted images locally on the device in user-selected directories.

1. **Login Security:**

* Implement lockout timer and self-destruction after multiple failed login attempts.

1. **Encryption History:**

* Maintain a secure local record of encrypted images and allow users to view and manage it.

### **2)** Non-Functional Requirements

1. **Security**

* All sensitive data (passwords, encryption keys) must be securely hashed or encrypted.
* App must not store plain-text passwords.

1. **Usability**

* The UI should be intuitive, minimal, and easy to use even for non-technical users.
* Real-time feedback should be provided during encryption/decryption processes.

1. **Performance**

* Image processing (encryption/decryption) should complete within a few seconds.
* App should remain responsive under normal device loads.

1. **Portability**

* The app should be compatible with Android smartphones.
* Should not rely on internet connectivity for core features.

1. **Reliability**

* The app must handle incorrect inputs and corrupted files gracefully.
* Biometric and password systems must be consistent and fail-safe.

1. **Data Privacy**

* No user data should be uploaded or shared externally.
* Encryption is done offline, ensuring complete local privacy.

## **Implementation**

This section presents each main page of the app, showing its purpose, and key features designed for secure and easy image encryption and decryption.

### **1)** Application Initialization Logic

Upon launching the application, the system determines whether user authentication credentials have been previously established by attempting to retrieve a hashed password from secure storage:

final storedPassword = await \_storage.read(key: 'appPassword');

* **If a password is detected**, the user is directed to the **Lock Screen** for authentication.
* **If no password is present** (indicating first-time use or reset), the user is redirected to the **Password Setup Screen** to configure credentials.

This decision flow is handled during the initState() lifecycle method of the LockScreen widget and ensures that unauthorized access is blocked from the outset.

### **2)** Password Setup Workflow

The **Password Setup Screen** enables the user to configure a secure password and optionally activate biometric authentication. The workflow includes:

* Password and confirmation fields with validation.
* Secure hashing of the password using the SHA-256 algorithm.
* Optional biometric enrollment through local\_auth, with device capability checks and dual biometric confirmation prompts.

**await \_storage.write(key: 'appPassword', value: \_hash(\_passCtrl.text));**

**await \_storage.write(key: 'biometricEnabled', value: \_bioEnabled.toString());**

In cases where a password has already been set, identity verification (via biometrics or password confirmation) is required before changes can be applied, providing an additional layer of protection.

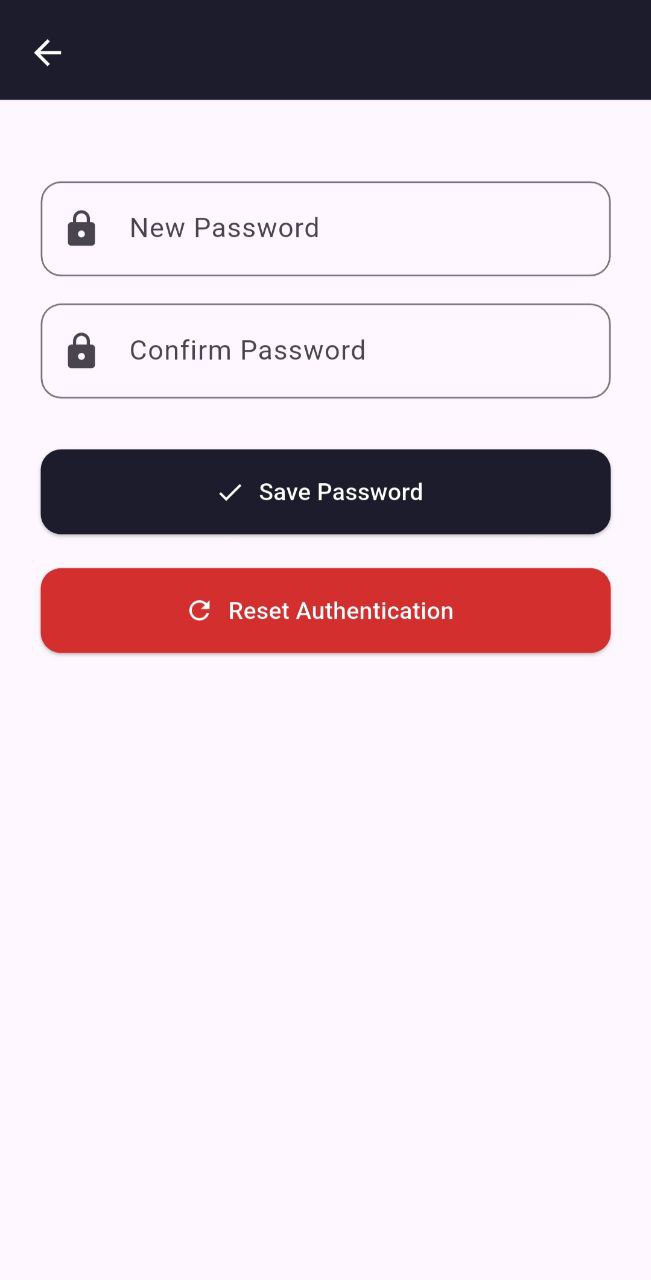


Figure1 : Create new Password Screen

### **3)** Lock Screen Authentication Logic

The **Lock Screen** serves as the primary authentication gate. It supports both password entry and biometric authentication (if previously configured). Key features include:

* **Password Verification**: User input is hashed and compared to the stored hash.
* **Biometric Login**: Leveraging platform biometrics (e.g., fingerprint, face ID) with fallbacks.
* **Failed Attempt Tracking**: Failed authentication attempts are persisted using secure storage.
* **Progressive Lockout**: Upon repeated failures, the user is temporarily locked out. Lockout duration increases with each threshold breach (e.g., 1 minute after 3 failures, 5 minutes after 6), deterring brute-force attempts.

**await \_storage.write(key: 'failedAttempts', value: \_failedAttempts.toString());**

* **Self-Destruct Mechanism**: After 9 failed attempts, the user is warned that one more failed attempt will result in full data erasure. If triggered, the application deletes all stored data and exits immediately to protect sensitive information.

**await \_storage.deleteAll();**

**exit(0)**

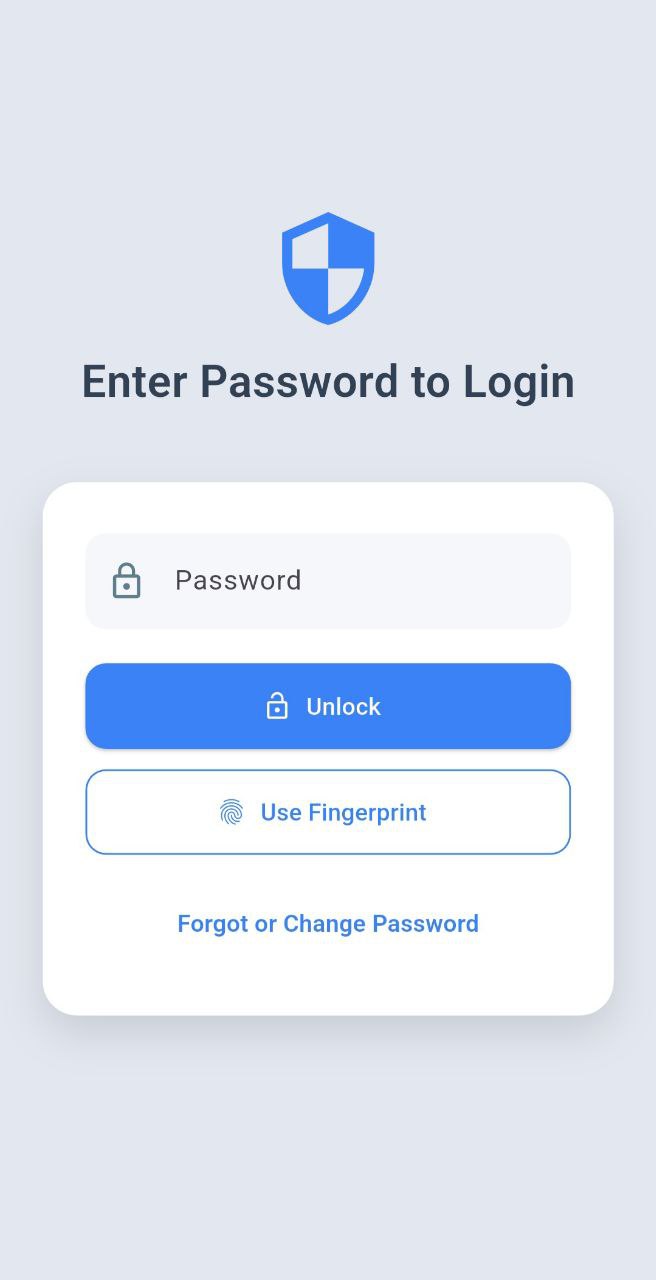


Figure2: Login Page

### **4)** Post-Authentication Navigation

Upon successful authentication **either via password or biometric validation** the user is granted access to the main interface of the app (MyHomePage), which is responsible for core application functionality.

**Navigator.pushReplacement(**

**context,**

**MaterialPageRoute(builder: (\_) => MyHomePage(title: 'Secure Your Memories')),**

**);**

This transition ensures a seamless and secure user experience while isolating authenticated content behind strong access controls.

### **5**) Home Page Functionality **(MyHomePage)**

Once the user has successfully authenticated, they are redirected to the MyHomePage, which serves as the central hub for interacting with secured images. The Home Page provides the following core features:

#### **5.1** Image Selection

Users can initiate the secure storage process by selecting an image from their device, either via:

* **Gallery Selection**
* **Camera Capture**

These options are presented in a bottom sheet modal using:

**\_showImageSelectionOptions() {**

**\_showBottomSheet([**

**BottomSheetOption(icon: Icons.photo\_library, label: "Choose from Gallery", onTap: () => \_handleImage(ImageController.pickFromGallery)),**

**BottomSheetOption(icon: Icons.camera\_alt\_rounded, label: "Take a Photo", onTap: () => \_handleImage(ImageController.captureFromCamera)),**

**]);**

**}**

Upon selection, the image is temporarily held in memory (\_image) and displayed within the UI using a ClipRRect container to maintain visual consistency.

#### **5.2** Image Encryption and Storage

With an image selected, users can encrypt and securely store it via:

**await ImageController.encryptAndSave(context, \_image);**

This operation uses the ImageController utility to handle encryption logic and file I/O securely. During encryption, a loading overlay is displayed to prevent user interaction and provide feedback.

If no image is selected, a warning snackbar is shown to prompt user action.

#### **5.3** Image Decryption

Users can also decrypt and load a previously saved encrypted image:

**final image = await ImageDecryptor.decryptAndReturnImage(context);**

This invokes a file picker to select an encrypted file, which is then decrypted and loaded into memory for viewing.

#### **5.4** Auxiliary Features

* **Fullscreen Image View**: Users can tap to open a full-screen, zoomable viewer for better inspection.
* **Image Clearing**: A button allows the current image to be removed from memory.
* **History Navigation**: A dedicated entry in the settings menu routes to a HistoryPage, allowing users to view previously encrypted/decrypted activity.
* **Logout Functionality**: Users can securely log out and return to the LockScreen using:

**Navigator.pushReplacement(context, MaterialPageRoute(builder: (\_) => LockScreen(onUnlocked: () {})));**

#### **5.5** UI and Interaction Design

The UI employs Flutter’s SafeArea, Stack, and SingleChildScrollView widgets to ensure responsiveness across device sizes. Custom buttons are styled consistently for primary (\_mainButton) and secondary (\_miniButton) actions, enhancing usability. Bottom sheet modals standardize user choices with a clean, tactile interface.

This implementation of the Home Page ensures both strong security and a user-centric experience, allowing for end-to-end control over image encryption workflows while maintaining performance and simplicity.

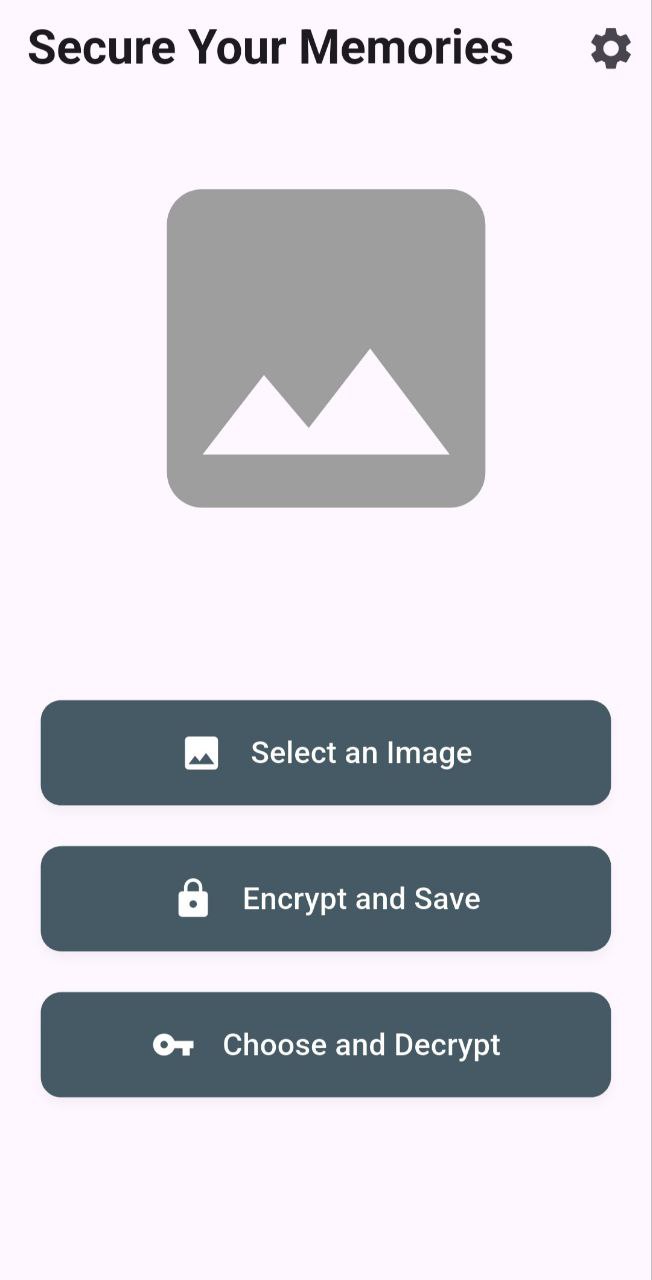


Figure3: Home Page Screen

### **6)** Password Input Page

The **Password Input Page** allows users to set a unique name and a strong password before encrypting an image. This step ensures each encrypted file is uniquely identifiable and protected by a secure credential.

#### Key Features:

* **Unique Image Naming:** Users must enter a distinct name for each image. The app prevents reuse by checking against previously stored names using SharedPreferences.
* **Strong Password Requirements:**

Passwords must meet the following criteria:

* At least 6 characters.
* One uppercase letter.
* One number.
* One special character.
* A real-time password strength indicator helps users choose secure passwords.
* **Confirmation and Validation:**  
  Password confirmation is required, and all inputs are validated dynamically. Only when all criteria are met does the “Encrypt” button become active.
* **Credential Reuse Prevention:**  
  Previously used image names and passwords are tracked locally to prevent duplicates and enhance security.
* **User-Friendly UI:**  
  The interface includes toggles for password visibility, clear error messages, and progress indicators, offering a smooth user experience.

#### Workflow:

1. Users input an image name and password.
2. The app checks for uniqueness and strength.
3. On successful validation, credentials are saved locally.
4. The data is returned to the previous screen to proceed with encryption.

This page strengthens image security by enforcing unique, strong credentials while maintaining a user-friendly experience.

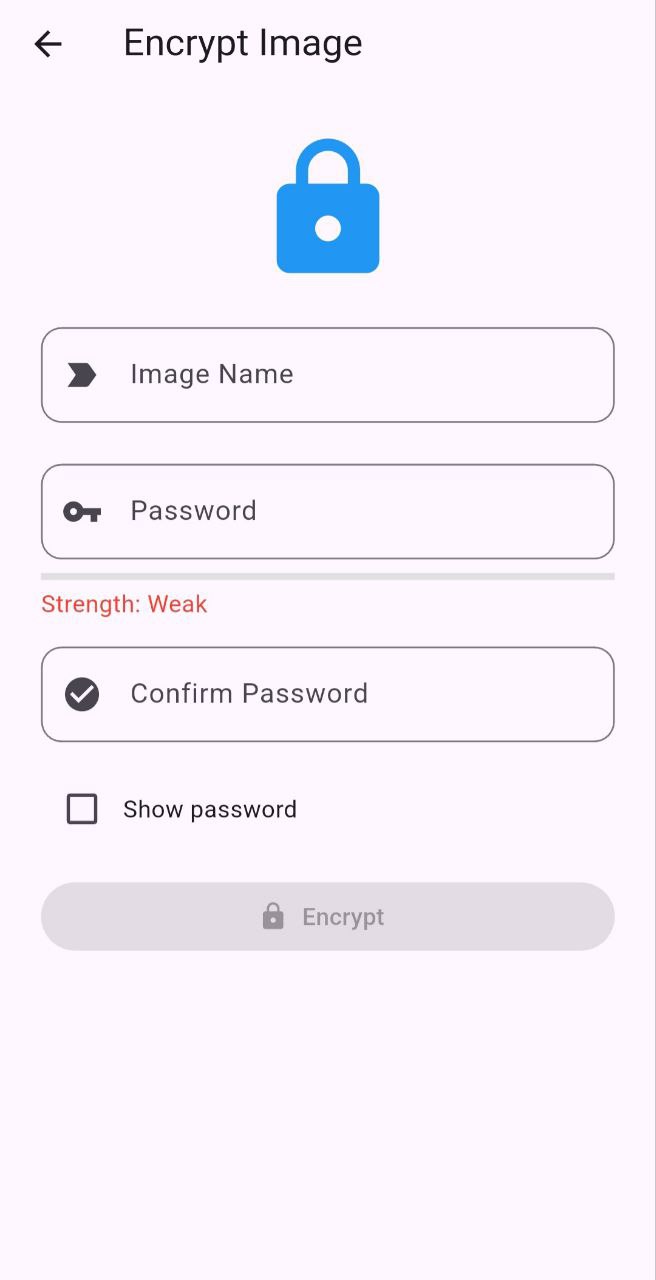


Figure4: Input Password Page

### **7**) Image Controller Logic

The **ImageController** class orchestrates the complete workflow for selecting, encrypting, and saving images securely. It acts as the core utility for integrating image input with the encryption pipeline, ensuring a smooth and secure user experience.

#### Key Responsibilities:

* **Image Acquisition:**  
  Users can select images via:
* **Gallery using ImagePicker.pickImage(source: ImageSource.gallery)**
* **Camera using ImagePicker.pickImage(source: ImageSource.camera)**
* **Encryption Workflow:**  
  The main method, encryptAndSave, handles encryption via the following steps:

**Credential Input:**  
Launches the PasswordInputPage where the user enters a unique image name and a strong password:

**final result = await Navigator.push<Map<String, String>>(context, MaterialPageRoute(builder: (\_) => const PasswordInputPage()));**

* **Folder Selection:**  
  Prompts the user to select a destination directory using:

**final folderPath = await FilePicker.platform.getDirectoryPath();**

* **Encryption Execution:**  
  Uses an ImageEncryptor instance to encrypt the image file with progress feedback shown through a ValueListenableBuilder:

**final savedPath = await encryptor.encryptImage(..., onProgress: (val) => progress.value = val);**

* **Secure History Logging:**  
  After successful encryption, the app stores a hashed version of the password and image metadata in SharedPreferences:

**final hashedPassword = sha256.convert(utf8.encode(password)).toString();**

**final newEntry = jsonEncode({...});**

**history.add(newEntry);**

**await prefs.setStringList('encryptionHistory', history);**

* **Error Handling and Feedback:**  
  Every stage includes user feedback via snackbars and modals, ensuring transparency and a responsive experience:
* **❌ Cancel or failure notifications.**
* **✅ Success messages with the encrypted file path.**

#### Summary:

The ImageController efficiently bridges the user interface and encryption engine, managing:

* Image selection
* User credential collection
* Destination path handling
* Secure AES encryption
* History tracking

It ensures that only valid, user-authorized images are encrypted, securely saved, and verifiably logged — aligning with the app’s overall emphasis on **security, usability, and accountability**.

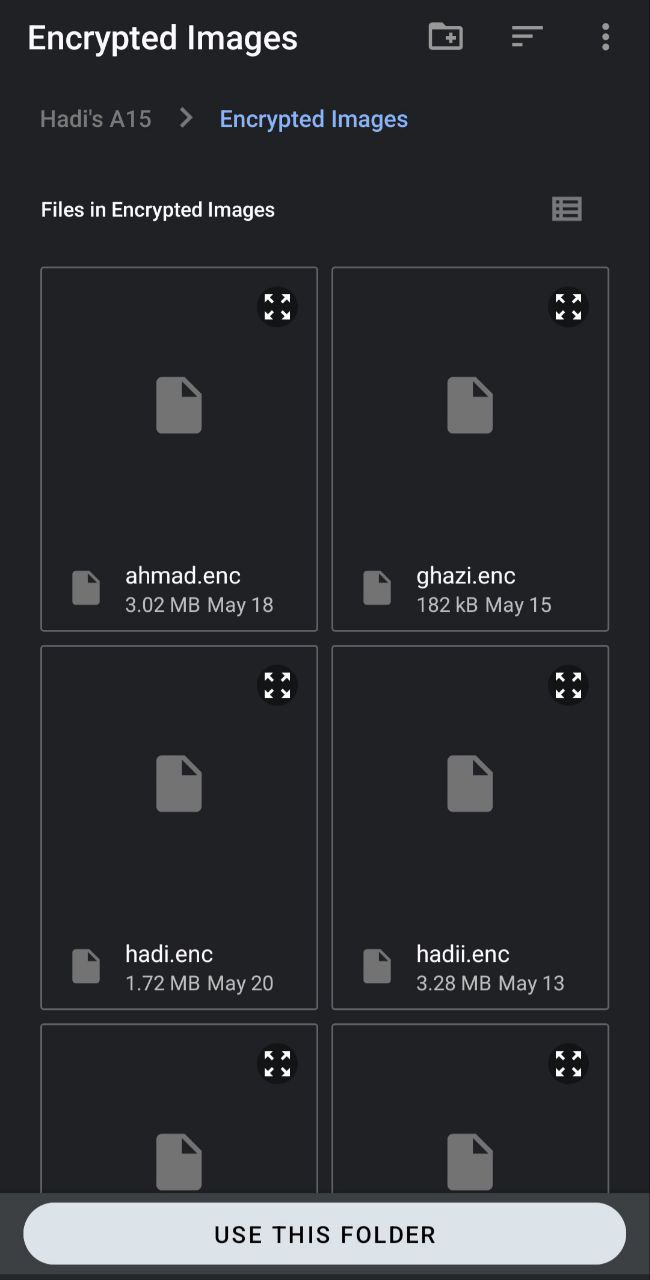


Figure5: Choose Folder

### **8**) Image Encryption Logic (ImageEncryptor)

The ImageEncryptor class handles the **core image encryption process** once a user selects a file destination. It ensures robust security by combining **AES-GCM encryption**, **PBKDF2 password derivation**, and a custom file format structure.

#### Key Features & Workflow

* **Storage Permission Check:**  
   On Android, it requests runtime permissions (Permission.storage, Permission, manage ExternalStorage) before proceeding:

**final storage = await Permission.storage.request();**

* **Secure Key Derivation:**  
  A cryptographic key is derived from the user’s password using **PBKDF2 with SHA-256**, 100,000 iterations, and a randomly generated salt:

**final key = await compute(\_deriveKeyInIsolate, {'password': password, 'salt': salt});**

* **AES-GCM Encryption in Isolate:**  
  The image is encrypted in a separate isolate for performance using **AES-256 GCM** mode, providing both confidentiality and integrity:

**final encrypted = await compute(\_encryptInIsolate, {...});**

* **Custom File Format for Encrypted Output:**  
  The resulting encrypted file includes:
* A **magic header** ("SC01") to identify valid encrypted files.
* The **salt** and **IV** (initialization vector).
* The **encrypted image bytes**.

**final output = BytesBuilder()**

**..add(utf8.encode(\_magicHeader))**

**..add(salt)**

**..add(iv)**

**..add(encrypted);**

* **Progress Feedback:**  
  Progress updates are sent to the UI using a **Function(double)? onProgress** callback to ensure smooth user experience **(e.g., onProgress?.call(0.7)).**

#### Summary

This encryption logic ensures high security by:

* Using **password-based key derivation** (PBKDF2 + SHA-256)
* Encrypting with **AES-GCM**, a modern authenticated encryption algorithm
* Running compute-heavy tasks off the main thread (via compute)
* Embedding all metadata (salt, IV, header) within the output file

This design guarantees that each encrypted image is unique, tamper-evident, and fully recoverable with the correct password.

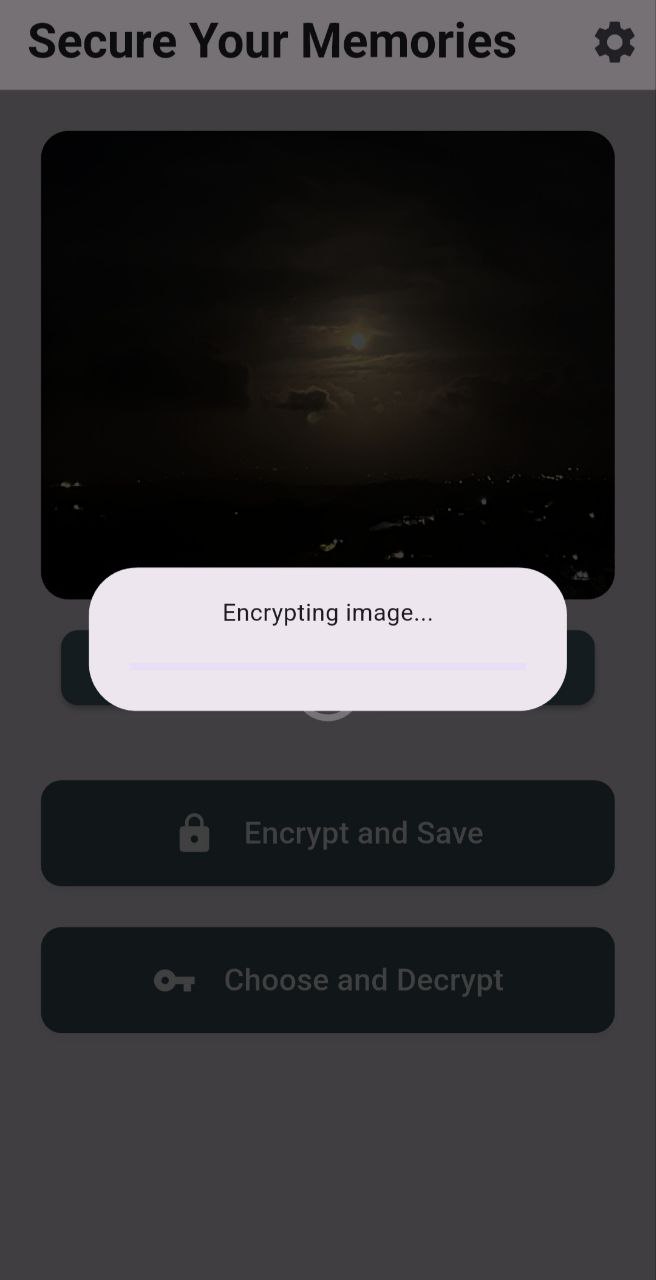


Figure6: Encrypting Process

### **9**) Image Decryption Logic (ImageDecryptor)

The ImageDecryptor class securely handles the process of **decrypting encrypted images** selected by the user. It ensures a robust experience with password validation, background processing, and user feedback.

#### Key Features & Workflow

* **Encrypted File Validation:**  
  Upon selecting a file via a file picker, the app checks for a custom header ("SC01") to confirm the file is in the correct encrypted format:

**if (utf8.decode(bytes.sublist(0, 4)) != \_header) {**

**\_showSnack(context, "❌ Not a valid encrypted file.");**

**return null;**

**}**

* **Password Confirmation Dialog:**  
  Before decryption begins, users must input and confirm their password in a secure form dialog:

**return showDialog<String>(...);**

* **Key Derivation & Decryption:**  
  The encrypted file contains the salt and IV. A 256-bit key is derived using PBKDF2 with SHA-256. Then, AES-GCM is used to decrypt the image:

**final key = derivator.process(...); // PBKDF2 key derivation**

**encrypter.decryptBytes(...); // AES-GCM decryption**

* **Decryption in Isolate:**  
   Heavy decryption logic runs in a separate isolate to keep the UI responsive:

**final decrypted = await compute(\_decryptInIsolate, {...});**

* **Temporary Output & Feedback:**  
  Once decrypted, the image is saved to a temporary directory and displayed to the user. If the password is incorrect, the user is notified and limited to 3 attempts before being blocked:

**if (attempts >= 3) await prefs.setBool('isBlocked', true);**

#### Summary

This decryption implementation ensures:

* Only valid, encrypted files are processed
* Strong password-based access using **PBKDF2 + AES-GCM**
* User-friendly UI with feedback dialogs and retry logic
* Decryption is securely isolated from the UI thread for performance

By combining cryptographic best practices and thoughtful UX design, the decryption flow complements the app's overall security-first image handling process.

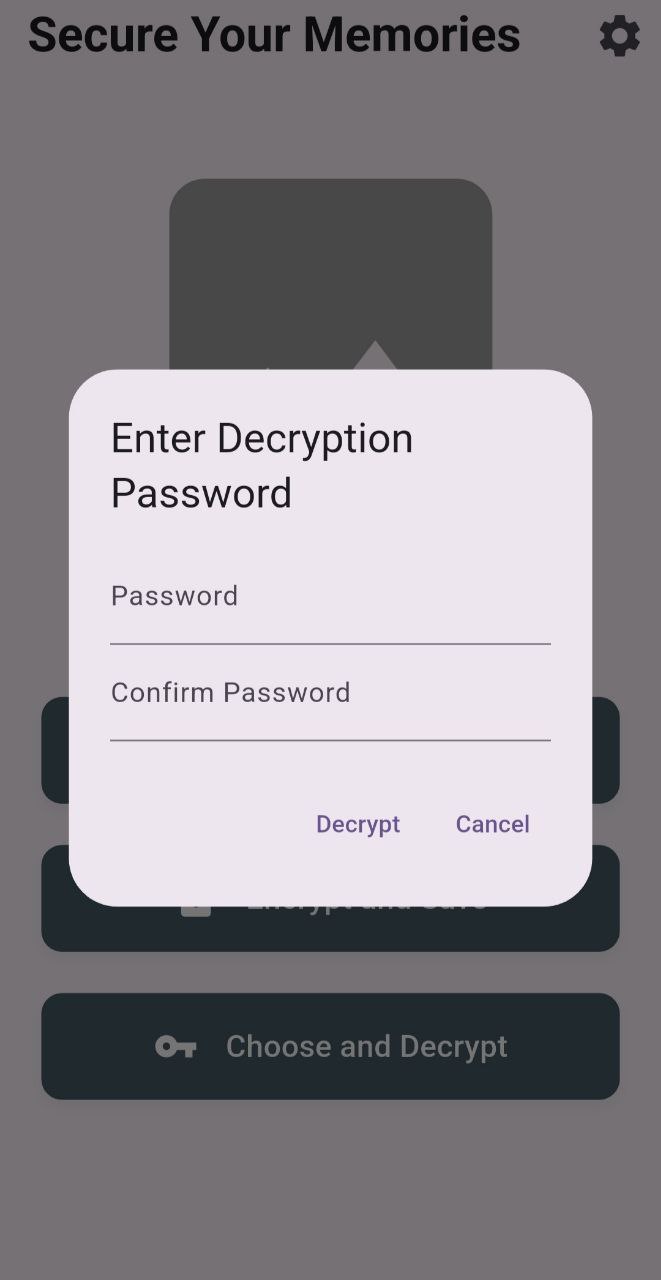


Figure7: Decrypting Password

### **10)** Encryption History Viewer (HistoryPage)

The History Page serves as a **user-friendly log** for tracking encrypted images and their metadata. It offers search, filtering, and management tools to enhance transparency and user control over stored activity.

#### Core Features

* **Automatic Loading of History:**  
   On initialization, the page retrieves a list of previous encryption events stored as JSON strings in SharedPreferences:

**final raw = prefs.getStringList('encryptionHistory') ?? [];**

**\_all = raw.map((e) => jsonDecode(e)).toList().reversed.toList();**

* **Real-Time Filtering:**  
  Users can filter records by:
* **Text Query (Image Name)**
* **Date Selection**  
  Updates occur instantly as users type or pick a date.
* **Interactive Record View:**  
  Each record shows:
  + Image name
  + Hashed password (with **long-press to copy**)
  + File path (trimmed to folder)
  + Timestamp
* **Record Management:**
  + **Individual Deletion:** Each entry includes a "Remove" button.
  + **Bulk Deletion:** The top-right delete icon clears all history after user confirmation.

#### UX and Design Considerations

* The UI is responsive and organized using cards for clean separation of entries.
* Filter buttons and snackbars provide intuitive, real-time feedback.
* Clipboard integration makes it easy to reuse password hashes if needed:

**Clipboard.setData(ClipboardData(text: text));**

#### Summary

The history page enhances usability and accountability by allowing users to review and manage encrypted image logs. It combines persistent storage, modern UI components, and practical features in a secure and efficient interface.

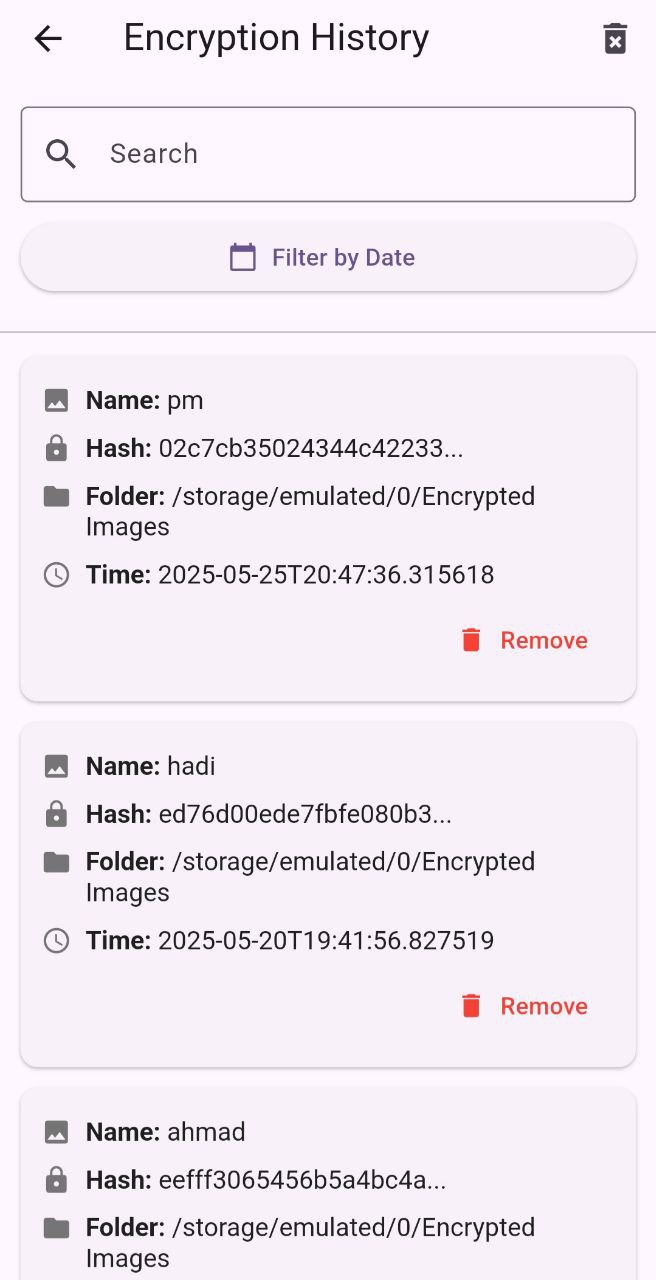


Figure8: History Page

### **11**) Permissions and Dependencies

#### Dependencies

#### The project utilizes a range of Flutter packages to enable image encryption, biometric authentication, and secure file handling. These are defined in the pubspec.yaml file under the dependencies section:

|  |  |
| --- | --- |
| **Package** | **purpose** |
| Image\_picker | |  | | --- | |  |  |  | | --- | | To capture or select images from the device. | |
| file\_picker | To browse directories and files, e.g., for saving encrypted images. |
| |  | | --- | |  |   encrypt   |  | | --- | |  | | Provides AES-GCM encryption and decryption capabilities. |
| shared\_preferences | Stores encryption history and user preferences locally. |
| path\_provider | |  | | --- | |  |  |  | | --- | | Provides access to directories for saving decrypted images. | |
| permission\_handler | Requests and manages runtime permissions like storage or camera. |
| flutter\_secure\_storage | |  | | --- | |  |  |  | | --- | | Offers encrypted local storage for future credential use (optional) | |
| image | Image processing for steganography or transformation if needed. |
| local\_auth | Enables biometric authentication using fingerprint or face recognition |
| cupertino\_icons | Provides iOS-style icons used in the UI. |

Tabel1: Table of Dependencies 1

These dependencies collectively enable the secure handling and storage of encrypted image files, as well as a polished and user-friendly experience.

#### Android Permissions

The app requests essential permissions in AndroidManifest.xml to perform encryption, decryption, and biometric validation:

| **Permission** | **Purpose** |
| --- | --- |
| READ\_MEDIA\_IMAGES | For Android 13+: Access to image files. |
| WRITE\_MEDIA\_IMAGES | Save modified/encrypted image files. |
| READ\_EXTERNAL\_STORAGE (maxSdk 32) | Read files on Android 12 and below. |
| WRITE\_EXTERNAL\_STORAGE (maxSdk 28) | Save files on legacy systems. |
| MANAGE\_EXTERNAL\_STORAGE | Full access to device storage on Android 11+ |
| CAMERA | Capture images from the device camera. |
| USE\_BIOMETRIC / USE\_FINGERPRINT | Authenticate users before decryption using biometrics. |

Tabel2: Tabel of Permissions 1

The use of tools:ignore="ScopedStorage" with MANAGE\_EXTERNAL\_STORAGE reflects deliberate intent to support unrestricted access to file directories for encryption operations, particularly when saving or retrieving .enc files.

# Chapter: 3

## Conclusion

This project successfully demonstrates the secure encryption and decryption of images using modern cryptographic techniques integrated within a user-friendly Flutter application. By combining file handling, AES encryption, biometric authentication, and persistent storage, the system ensures both data protection and usability. It offers a practical and efficient solution for safeguarding visual data, marking a step forward in privacy-aware mobile development. The implementation reflects strong attention to detail, reliability, and user experience, making it suitable for real-world applications where image confidentiality is critical.

## **Future Work**

To enhance the app's functionality and broaden its scope, several key features are planned:

1. **Batch Encryption** – Support for encrypting multiple images at once to improve efficiency.
2. **Video Encryption** – Extend capabilities to include secure encryption of video files.
3. **Steganography Integration** – Embed encrypted data within images for added secrecy and layered protection.
4. **Cross-Platform Enhancements** – Improve performance and permission handling across Android, iOS, and future platforms.
5. **Secure Cloud Sync** – Enable encrypted backup and access across trusted devices.

**"Thank you for your time, attention, and support throughout this project your guidance and interest have been invaluable to its success."**