TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

2025-2026

## Cybersecurity Project Report

---

# Secure Laboratory Image Encryption

*A Secure Messaging Solution for Laboratory Diagnostics*

---

MEDICRYPTIS


## Presented by:

Sabrine Ayadi
Aya Hafsi
Sayda Ouaddar
Mayssam Hassen

## Supervised by:

Mr. Manel Abdelkader

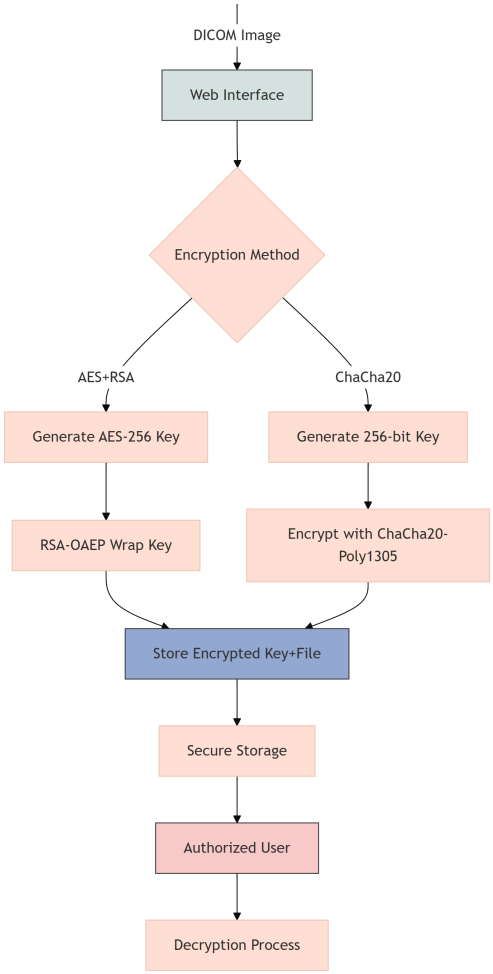

---

# Contents

# 1 Workflow

**Users log in with role-based access as a doctor, technician, admin, or patient. They upload medical images (DICOM, PNG, or JPEG) via drag-and-drop, and the system automatically extracts metadata like patient ID and scan date. Users then select an encryption method: fast ChaCha20 or hybrid AES + RSA, prompting the system to generate encryption keys.**

**The image is encrypted accordingly and optionally pixelated for added privacy. All encrypted data and metadata are then securely stored, with key handling managed by a Key Management System (KMS). Every action is tracked through audit logging. When needed, authorized users can request access;. after verification, the system decrypts the key using their RSA private key, and the image is revealed for viewing.**



# 2 Components

The architecture is divided into three parts:

## 2.1 Client-Side (User Side) Components

### a. User Interface (UI)

- Displays different views based on the user's role (Admin, Technician, Researcher).
- Allows image upload through a drag-and-drop feature.
- Provides an option to choose the encryption method: ChaCha20 or AES+RSA.

### b. Encryption Module

- **ChaCha20:** Uses a 256-bit key and a 96-bit nonce. It mixes the keystream with image data to produce a secure, encrypted version.
- **AES+RSA:** First, the image is encrypted using AES. Then, the AES key is encrypted using RSA to ensure secure key exchange.

## 2.2 Server-Side Components

### a. Authentication and Access Control

- Enforces login and role verification.
- Only allows authorized users to decrypt or access images.

### b. Key Management System (KMS)

- Generates a unique AES key for every uploaded image.
- Stores RSA keys in a secure environment.
- Applies regular key rotation for enhanced security.

### c. Secure Storage Server

- Stores encrypted images and associated metadata in a secure and tamper-proof way.

## 2.3 Integration Components

### a. API Gateway

- Acts as a bridge between the frontend and backend.

- Provides the following endpoints:
  - `/upload` – to save and encrypt an image.
  - `/decrypt` – to retrieve and decrypt an image.

**b. Transmission Protocol**

- Uses HTTPS with TLS 1.3 to ensure secure data transmission between client and server.

# 3 ROLES/USERS

**User Roles and Descriptions**

| User | Role | Description |
|---|---|---|
| **Laboratory Technician** | Encryptor & Sender | Receives medical image/test results from lab equipment and encrypts them. Sends the encrypted image and encrypted key to both the treating doctor and the client (patient). The message is sent via email or a message on the platform. The receiver will have a specific kind of encryption. |
| **Treating Doctor** | Authorized Decryptor/encypter | Receives the encrypted image and uses their private generated key to decrypt it. Uses the result for diagnosis and treatment. The doctor can also re-encrypt the image to resend it to the laboratory technician or the client. |
| **Patient / Client** | Authorized Decryptor | Has the right to access their own medical results. Receives the encrypted image and uses the provided key to decrypt and view the result. |

# 4 Message Exchange

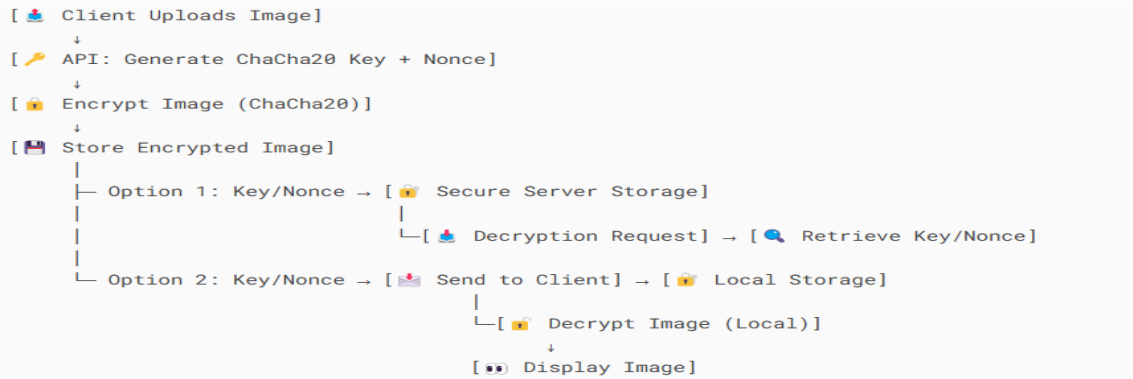
## 4.1 AES Encryption Flow

Table 1: **AES Encryption Flow**

| Message/Data | Direction | Description |
|---|---|---|
| Image Upload | Client to API | Raw image uploaded by the user to be encrypted. |
| AES Key Generation | API to Encryption Engine | Backend generates a random AES key to encrypt the image. |
| Image Encryption | Encryption Engine to API | Image is encrypted using AES. |
| AES Key Encryption | Encryption Engine to Key Manager to Client | AES key is encrypted using the user's RSA public key and stored/transmitted. |
| Encrypted Image Storage | API to Storage | The encrypted image is securely stored in the storage system. |
| Decryption Request | Client to API | Authorized user requests decryption of a specific image. |
| AES Key Retrieval | Client to Key Manager | Encrypted AES key is retrieved by the client. |
| AES Key Decryption | Client (Local) | Client uses their RSA private key to decrypt the AES key locally. |
| Image Decryption | Client (Local) | Image is decrypted using the decrypted AES key. |
| Decrypted Image Display | Client | Final decrypted image is shown or downloaded by the user. |


## 4.2 ChaCha20 Encryption Flow

| Message/Data | Direction | Description |
|---|---|---|
| Image Upload | Client to API | Raw image uploaded by the user to be encrypted. |
| ChaCha20 Key Generation | API to Encryption Engine | Backend generates a random 256-bit ChaCha20 key and nonce. |
| Image Encryption | Encryption Engine to API | Image is encrypted using ChaCha20 with the generated key. |
| Encrypted Image Storage | API to Storage | The ChaCha20-encrypted image is securely stored. |
| Decryption Request | Client to API | Authorized user requests decryption of a specific image. |
| ChaCha20 Key Retrieval | Client to Key Manager | Encrypted ChaCha20 key is retrieved (if not already locally stored). |
| Image Decryption | Client (Local) | Image is decrypted locally using the ChaCha20 key and nonce. |
| Decrypted Image Display | Client | Final decrypted image is shown or downloaded by the user. |




# 5 Tools and Technologies

## 5.1 Backend Programming Language

we developped the backend using Python, due to its simplicity and the availability of cryptographic libraries. The web application is built using the Flask framework, which will help manage routing, handle user sessions, and structure the project efficiently

## 5.2 Encryption Libraries

For encryption, we used PyCryptodome, a Python library that supports secure and efficient cryptographic operations. we implemented AES for symmetric encryption and RSA for key exchange in a hybrid setup. In addition, we implemented the ChaCha20 stream cipher as an alternative encryption method to give users a choice between speed and structure.

## 5.3 Image Processing Tools

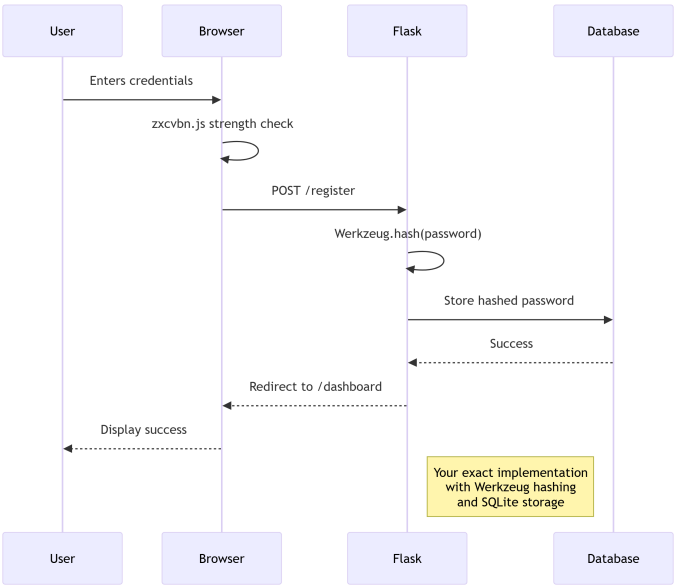
To visually obscure encrypted images, we used OpenCV to apply a pixelation effect. This will act as an additional layer of visual obfuscation, even though it is not a cryptographic method. It will help hide content during encrypted preview stages.

## 5.4 Frontend Technologies

On the frontend, we used HTML5, CSS3, and Bootstrap to design a clean and responsive interface. The app will adopt a dark theme with design elements that reflect a cryptography and lab-based aesthetic.

## 5.5 Password Strength Validation

For enhanced user security, I will incorporate zxcvbn.js, a JavaScript password strength estimator, into the registration form. It will give real-time feedback and discourage weak password choices.




### 5.6 Version Control Hosting

**After development and testing, we will deploy the app on platforms for public access.**

## 6 Development Phases

### 6.1 Requirements Gathering

**we began by clearly identifying the user requirements and the expected flow of the system. The application must support image upload, encryption using either AES+RSA or ChaCha20, pixelation, and image decryption ,all while maintaining a secure login system for users.**

### 6.2 System Design

**As for the system design architecture, we defined the structure of the application, the data flow, and the key modules involved. we mapped out routes such as:**

- /register and /login for user authentication
- /encrypt and /decrypt for image processing
- /download for result retrieval

### 6.3 Frontend Implementation

**we started by building the user-facing pages: login, registration, and dashboard. we integrated zxcvbn.js into the registration form for password strength validation. The dashboard will allow users to:**

- Upload images
- Choose an encryption method
- View the pixelated encrypted result
- Decrypt their images

**The design will follow a consistent dark theme and use Bootstrap for layout and responsiveness.**

## 6.4 Backend and Core Features

**we implementd the encryption logic using PyCryptodome. For AES+RSA, we encrypted the image with a randomly generated AES key and protected that key using RSA. For ChaCha20, we generated the key and nonce programmatically and used them to encrypt the image stream.**

**Once encryption is complete, we applied a pixelation effect using OpenCV. All logic is modular to allow easy switching between encryption methods.**

## 6.5 Database and User Management

**we set up an SQLite database to manage user data. Passwords are hashed securely using Werkzeug's utilities, and login sessions are managed by Flask. we ensured the system protects against unauthorized access and injection attacks.**

## 6.6 Testing and Debugging

**After building the core components, we conducted extensive testing. we uploaded images of different sizes and formats to test compatibility. we verified that both encryption methods produce correct, reversible outputs and that pixelation works across all cases.**