

ABSTRACT

Among many that are taking the establishment of electronic personal health records for patients seriously, the ones hampered by environments that are too flexible and accessible for data outsourcing have included cloud computing. It is into these environments that patients desire to be granted ownership so as to manage their health in a resilient and scalable manner. For these very PHRs, which harbor patently sensitive information, security and privacy thus very much loom large in the foreground. Sufficiently, PHR owners should be in a position to flexibly and securely define their access policies on the outsourced data. Normally, commercial cloud platforms maintain confidentiality through symmetric or public key cryptography. In their traditional sense, these have for the most part been assumed within the normal, classic, sense; however, some of these considerations would not bear upon the outsourced data installations, due to various management overheads in key management and prohibitive maintenance costs entailed by management of scores of ciphertext copies. This paper is an attempt to construct and develop a fine-grained secure access control scheme for lightweight modification of access policies over private outsourced PHRs. This would add to existing services provided by CP-ABE and PRE. A notable contribution made in this paper is the proposal for policy versioning to enable complete traceability of policy changes. The paper concludes with a performance evaluation to demonstrate the efficiency of the proposed scheme.

Keywords: PHRs, access control, CP-ABE, policy update, proxy re-encryption, policy versioning, SSpformance evaluation.

INDEX

Contents

<u>CHAPTER 1 – INTRODUCTION</u>	3
<u>CHAPTER 3 – FEASIBILITY STUDY</u>	5
<u>a.Technical Feasibitiy</u>	5
<u>b.Operational Feasibility</u>	5
<u>CHAPTER 4 – SYSTEM REQUIREMENT SPECIFICATION DOCUMENT</u>	7
<u>a.Overview</u>	7
<u>b.Module Description</u>	7
<u>c.Process Flow</u>	7
<u>d.SDLC Methodology</u>	9
<u>e.Software Requirements</u>	13
<u>f.Hardware Requirements</u>	13
<u>CHAPTER 5 – SYSTEM DESIGN</u>	14
<u>a.DFD</u>	14
<u>b.ER diagram</u>	15
<u>c.UML</u>	16
<u>d.Data Dictionary</u>	22
<u>CHAPTER 6-TECHNOLOGY DESCRIPTION</u>	27
<u>CHAPTER 7 – TESTING & DEBUGGING TECHNIQUES</u>	29
<u>CHAPTER 8 – OUTPUT SCREENS</u>	31
<u>CHAPTER 9 -CODE</u>	43
<u>CHAPTER 10 – CONCLUSION</u>	58
<u>CHAPTER 11 – BIBLOGRAPHY</u>	59

CHAPTER 1 – INTRODUCTION

In outsourced data sharing environments like cloud storage systems, ensuring continuous availability and secure access to shared data is essential. Many organizations prefer cloud storage for its cost-effectiveness and efficient resource management. To protect sensitive data, owners typically encrypt it before outsourcing; however, encryption alone is not sufficient. Fine-grained access control is also necessary, and Ciphertext-Policy Attribute-Based Encryption (CP-ABE) is commonly used for this purpose. CP-ABE enables "one-to-many" encryption with access policies defined by the data owner, providing both encryption and access control capabilities. Despite its advantages, CP-ABE incurs high overhead during attribute revocation or policy updates, requiring costly re-encryption and key redistribution—especially challenging in systems with many users. To address these issues, we propose a model that combines symmetric encryption for data (to enhance performance) with CP-ABE to encrypt only the symmetric key, thereby limiting the impact of policy changes. Additionally, we introduce a proxy re-encryption (PRE) mechanism that offloads re-encryption tasks from the data owner to a proxy, reducing computational and communication costs. We also implement a policy versioning system that records updates for future reference and auditing, and we leverage parallel programming to speed up cryptographic operations. Our solution enhances access control efficiency in multi-authority cloud environments, particularly in scenarios like Personal Health Record (PHR) sharing, and we provide security and performance evaluations to demonstrate its practical viability.

CHAPTER 2 – SYSTEM ANALYSIS

a. Existing system

Traditional systems for managing Personal Health Records (PHRs) in the cloud use symmetric or public key encryption to protect data. However, these methods are not well-suited for cloud environments where data is accessed by multiple users. They require managing many encryption keys or generating multiple copies of encrypted data, which increases storage and computational costs. These systems also lack fine-grained access control, making it hard to restrict data based on roles or attributes (e.g., doctor, emergency case). Updating access policies is inefficient, often requiring the data owner to re-encrypt all data manually. Moreover, there is no support for tracking policy changes, which affects transparency and auditability. In short, existing systems are not flexible, scalable, or efficient enough for dynamic and secure PHR sharing in modern cloud-based healthcare environments.

DISADVANTAGES:

- ✓ **Lack of Fine-Grained Access Control:** Cannot restrict access based on specific user attributes like role, department, or condition (e.g., emergency).
- ✓ **High Computational Overhead:** requires multiple re-encryptions and ciphertext copies, consuming more storage and processing power.
- ✓ **Inefficient Policy Updates:** Any change in access policy forces the data owner to re-encrypt the entire dataset, which is time-consuming.
- ✓ **Poor Scalability:** Struggles to handle large numbers of users and frequent access policy changes in a cloud environment.
- ✓ **No Policy Traceability:** Lacks version control or audit trail of access policy changes, reducing transparency and trust.
- ✓ **Complex Key Management:** Symmetric and public-key cryptography require handling numerous keys, which complicates system maintenance.

b. Proposed system

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

The proposed system offers a secure, flexible, and scalable framework for managing access to outsourced Personal Health Records (PHRs) by integrating Ciphertext-Policy Attribute-Based Encryption (CP-ABE), Proxy Re-Encryption (PRE), and policy versioning. CP-ABE enables data owners to define fine-grained access control policies based on specific user attributes, ensuring that only authorized users can decrypt and access sensitive health information. To reduce the computational burden on data owners, the system employs PRE, allowing a trusted proxy server to handle re-encryption tasks whenever access policies are updated. This eliminates the need for repeated manual re-encryption by the owner. Additionally, the system introduces policy versioning, which keeps a detailed history of all policy changes, enabling rollback and auditability. This feature enhances transparency and security while maintaining control over data access. The overall design ensures strong privacy protection, efficient policy updates, and scalability, making it highly suitable for secure and dynamic PHR sharing in modern cloud-based healthcare environments.

ADVANTAGES:

- **Enhanced Security:** The proposed system addresses vulnerabilities in traditional encryption methods by integrating CP-ABE and PRE, offering strong protection against unauthorized access and enabling secure policy updates.
- **Fine-Grained Access Control:** Using CP-ABE, the system allows access to data based on user attributes (e.g., role, department), ensuring only specifically authorized users can decrypt sensitive health records.
- **Reduced Computational Burden:** Proxy Re-Encryption (PRE) enables secure re-encryption of data without involving the data owner, minimizing the computational overhead on user devices.
- **Efficient Policy Updates:** Access policies can be updated dynamically without requiring full data re-encryption, making the system responsive to changing authorization needs.

CHAPTER 3 – FEASIBILITY STUDY

- **Technical Feasibility**

The proposed system is technically feasible as it uses established cryptographic techniques like Ciphertext-Policy Attribute-Based Encryption (CP-ABE) and Proxy Re-Encryption (PRE), both of which are well-supported by existing libraries in languages such as Python and Java. CP-ABE enables fine-grained access control based on user attributes, while PRE offloads re-encryption tasks to a proxy server, reducing the computational load on data owners. Policy versioning can be implemented using standard database techniques, allowing tracking and rollback of access policy changes. The system can be deployed on popular cloud platforms like AWS or Azure, which provide secure storage, scalability, and high availability. Additionally, the architecture supports integration with existing healthcare systems through RESTful APIs and modular components. The required tools, such as key management, encryption/decryption modules, and access verification, are readily available, making the system practical for real-world deployment.

- **Operational Feasibility**

The proposed system is operationally feasible as it is designed to align with the workflows of patients, healthcare providers, and cloud service operators. The system provides an intuitive interface for data owners (patients) to define and update access policies without requiring deep technical knowledge. By automating complex encryption processes like CP-ABE and re-encryption via a proxy server, the system ensures smooth operation with minimal user intervention. Healthcare professionals can securely access patient data based on their roles or attributes, improving coordination and decision-making without compromising data privacy. The inclusion of policy versioning further enhances usability by enabling easy tracking of access changes, which is essential in real-world healthcare settings where access requirements frequently evolve. The system can be integrated into existing hospital management or electronic health record systems using APIs, ensuring it fits well into current infrastructure without major operational disruptions. Cloud-based deployment ensures high availability,

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

scalability, and easy maintenance, making it suitable for both small clinics and large healthcare institutions. Overall, the proposed solution enhances operational efficiency by reducing manual tasks, simplifying access control, and ensuring secure, compliant sharing of sensitive health data in daily use.

- **Economic Feasibility**

The proposed system is economically feasible as it minimizes both development and operational costs while delivering high-value security and access control for Personal Health Records (PHRs). By using open-source cryptographic libraries for implementing Ciphertext-Policy Attribute-Based Encryption (CP-ABE) and Proxy Re-Encryption (PRE), the project avoids expensive licensing fees and reduces the overall development budget. Cloud deployment allows for flexible cost models, such as pay-as-you-go pricing offered by platforms like AWS, Azure, or Google Cloud. This eliminates the need for heavy investment in physical infrastructure and allows the system to scale economically based on usage. Operational costs are also reduced through automation. By offloading re-encryption tasks to a proxy server and allowing easy policy updates, the system reduces the time and effort required by administrators and data owners. This lowers human resource costs and enhances productivity. Additionally, by enhancing data security and access control, the system helps avoid potential financial losses due to data breaches, legal penalties, and non-compliance with healthcare regulations. The return on investment (ROI) is high, considering the long-term benefits in terms of data privacy, scalability, and secure health data sharing.

CHAPTER 4 – SYSTEM REQUIREMENT SPECIFICATION DOCUMENT

- **Overview**

The proposed system is developed using Python 3.6+ as the core programming language and is implemented within the PyCharm Integrated Development Environment (IDE). It utilizes the Django web framework to build the backend logic and manage server-side functionalities. On the client side, the user interface is designed using standard web technologies including HTML, CSS, Bootstrap, and JavaScript to ensure responsiveness and user-friendliness. For data handling and processing, essential libraries such as Pandas, Os, and Smtplib are employed. The system uses SQLite as the primary database for storing user information, encrypted PHRs, and access policies, offering a lightweight and file-based solution ideal for rapid development and deployment. Server deployment is managed using the XAMPP server environment to simulate web hosting locally during testing and demonstration phases. The entire system is compatible with Windows 7, 8, and 10 operating systems, making it widely accessible on standard machines. This combination of tools and technologies ensures the development of a secure, efficient, and easily maintainable platform for managing personal health records with fine-grained access control.

- **Module Description**

The proposed system comprises multiple interlinked modules, which are dedicated to delivering secure and efficient management of personal health records (PHRs) in the environment of cloud computing. Each module handles its task: data encryption and policy management as well as access and data-sharing functionalities. The following deals with the major modules of the system:

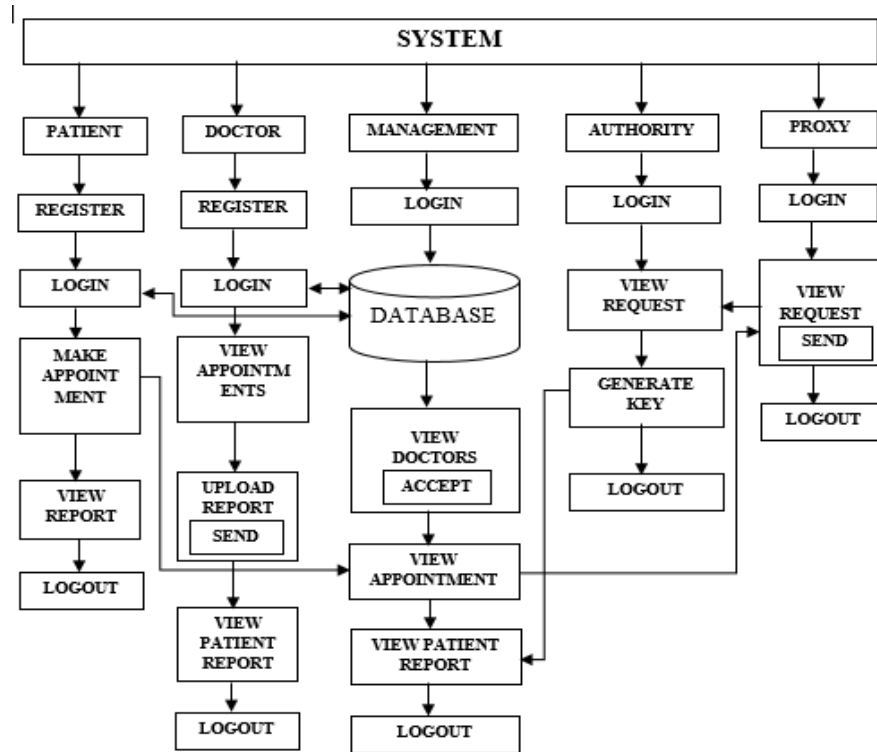
- **PATIENT:**
 - **Login:** Patient has to login with valid details which are used in his / her Registration.
 - **Register:** Each and every patient has to register.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **Appointment:** Patient will raise an appointment with symptoms.
- **View Report:** Patient will view reports after his / her medical checkup. Medical Records of every Hospitals must be visible using there generated key.
- **View health history:** patient can view their previous health data.
- **Logout:** Finally, logout from the system.
- **DOCTOR:**
 - **Login:** Doctor has to login with valid details which are used in his / her Registration
 - **Register:** Each and every patient has to register and management has to accept request.
 - **View Appointment:** View all the appointments
 - **Upload Report:** Uploads report.
 - **Send File:** Send's file to the proxy
 - **View Patient Report:** patient will view all the reports of the patient.
 - **Logout:** Finally, logout from the system.
- **HOSPITAL MANAGEMENT:**
 - **Login:** Management will login with default details, they can view the patients' medical details without any key.
 - **Appointment:** Views all the appointments requests from the Patients
 - **View Doctor Request:** View all the doctor requests who are registered.
 - **Send Information:** Patient request will be passed to doctor
 - **View Report:** View Patient Report in Emergency cases.
 - **Logout:** Finally logout from the system.
- **AUTHORITY:**
 - **Login:** Authority will login with default details
 - **View Request:** View all requests from proxy.
 - **Generate Key:** Generate Key to pass it to the authorized patient
 - **Logout:** Finally logout from the system.
- **PROXY SERVER:**
 - **Login:** Authority will login with default details
 - **View Request:** View all requests from Doctors.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **Send Request:** Pass requests to the authority
- **Process Flow**



- **SDLC Methodology**

The Software Development Life Cycle (SDLC) methodology defines the structured process for developing the **Secure and Scalable Access Control Framework for Outsourced PHRs Using CP-ABE, Proxy Re-Encryption, and Policy Versioning**. This methodology ensures a systematic approach from requirement analysis to deployment and maintenance. For this project, the **Waterfall SDLC methodology** is followed due to its clear, sequential phases where each step builds on the completion of the previous one.

1. Requirement Gathering & Analysis

In this initial phase, a thorough analysis of the challenges in existing PHR access control systems was conducted, particularly those related to dynamic access management and secure data outsourcing. Key technologies such as Ciphertext-Policy Attribute-Based Encryption (CP-ABE), Proxy Re-Encryption (PRE), and policy versioning were selected to address these challenges. Specific functional and non-functional requirements were defined, including secure encryption, access control enforcement, efficient policy updates, and traceability.

- **Outcome:** A well-defined set of system requirements focusing on scalability, flexibility, and security of health data access in cloud environments.

2. System Design

Based on the requirements, the system architecture was designed to include modules for encryption/decryption, policy generation, proxy re-encryption, and policy versioning. The design specified how users interact with the system (patients and healthcare providers), how data is securely stored and shared, and how policy changes are managed. Supporting diagrams such as Data Flow Diagrams (DFDs), UML class diagrams, and system architecture flowcharts were prepared.

- **Outcome:** Detailed system design for implementing secure, attribute-based access control with support for dynamic updates and version control.

3. Implementation

During this phase, the system was developed using **Python**, with **Django** as the web framework. The encryption logic (CP-ABE and PRE) was implemented along with modules for managing users, uploading PHRs, defining access policies, and performing re-encryption. Policy versioning was integrated using SQLite as the backend database. The user interface was created using HTML, CSS, Bootstrap, and JavaScript.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **Outcome:** A fully functional system capable of securely encrypting and managing PHRs with dynamic policy control.

4. Testing

The system underwent extensive testing to ensure correctness, security, and performance. Functional testing validated that only authorized users could access encrypted data based on defined attributes. Policy updates and versioning features were tested for traceability and rollback functionality. Proxy re-encryption was tested to ensure that it correctly offloaded computational load without compromising security.

- **Outcome:** A validated system with successful enforcement of access control policies, secure encryption mechanisms, and efficient policy management.

5. Deployment

Since the system is designed for cloud-based environments, it was deployed locally using **XAMPP Server** for simulation. The deployment included setup of the Django backend, SQLite database, and hosting the application with user interfaces for patients and healthcare providers. The application is ready for future deployment to cloud servers such as AWS or Azure for real-world use.

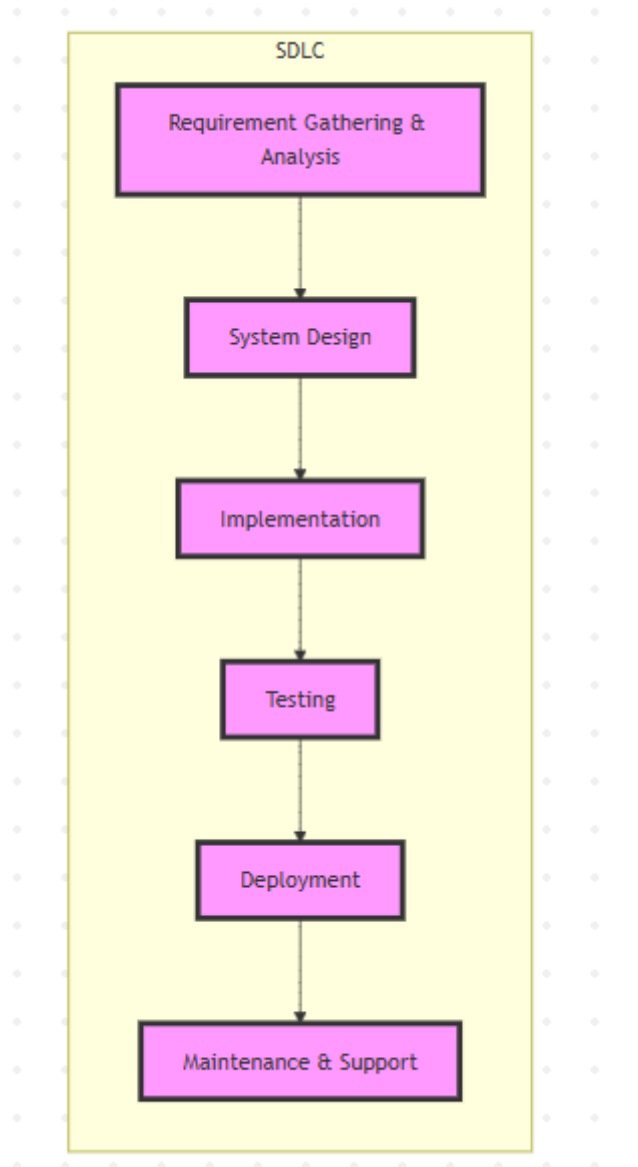
- **Outcome:** A deployable web-based application simulating secure PHR sharing and policy enforcement in a cloud environment.

6. Maintenance & Support

Post-deployment, the system is designed for ongoing enhancement. New features such as role-based dashboards, real-time policy notifications, or integration with Electronic Health Record (EHR) systems can be added. Maintenance also includes applying updates based on user feedback, security patches, and adapting to evolving cryptographic standards.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **Outcome:** A scalable and maintainable access control system for outsourced PHRs, capable of future upgrades and real-world integration.



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **Software Requirements**

Operating System	: Windows 7/8/10
Server side Script	: HTML, CSS, Bootstrap & JS
Programming Language	: Python
Libraries	: Flask or Django, Pandas, SQLite, Os, Smtplib
IDE/Workbench	: PyCharm
Technology	: Python 3.6+
Server Deployment	: Xampp Server
Database	: SQLite

- **Hardware Requirements**

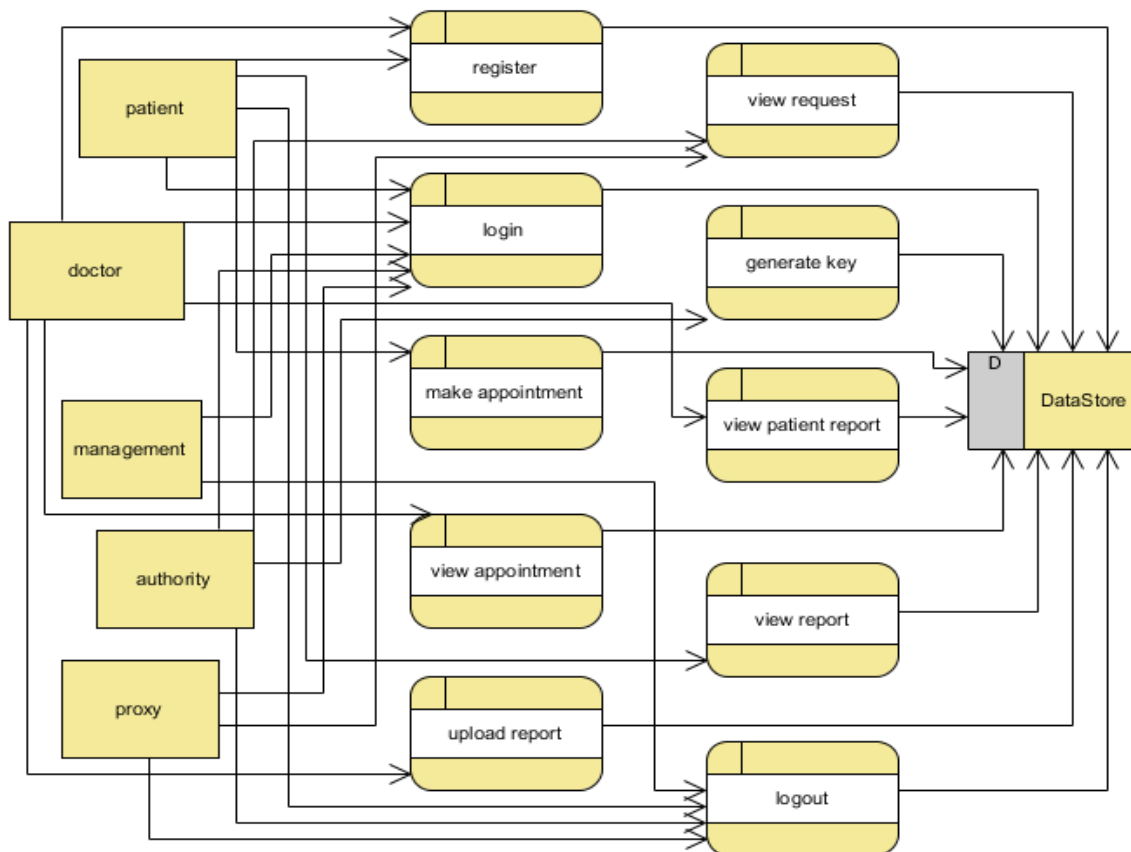
Processor	- I3/Intel Processor
Hard Disk	- 160GB
Key Board	- Standard Windows Keyboard
Mouse	- Two or Three Button Mouse
Monitor	- SVGA
RAM	- 8GB

CHAPTER 5 – SYSTEM DESIGN

a. DFD

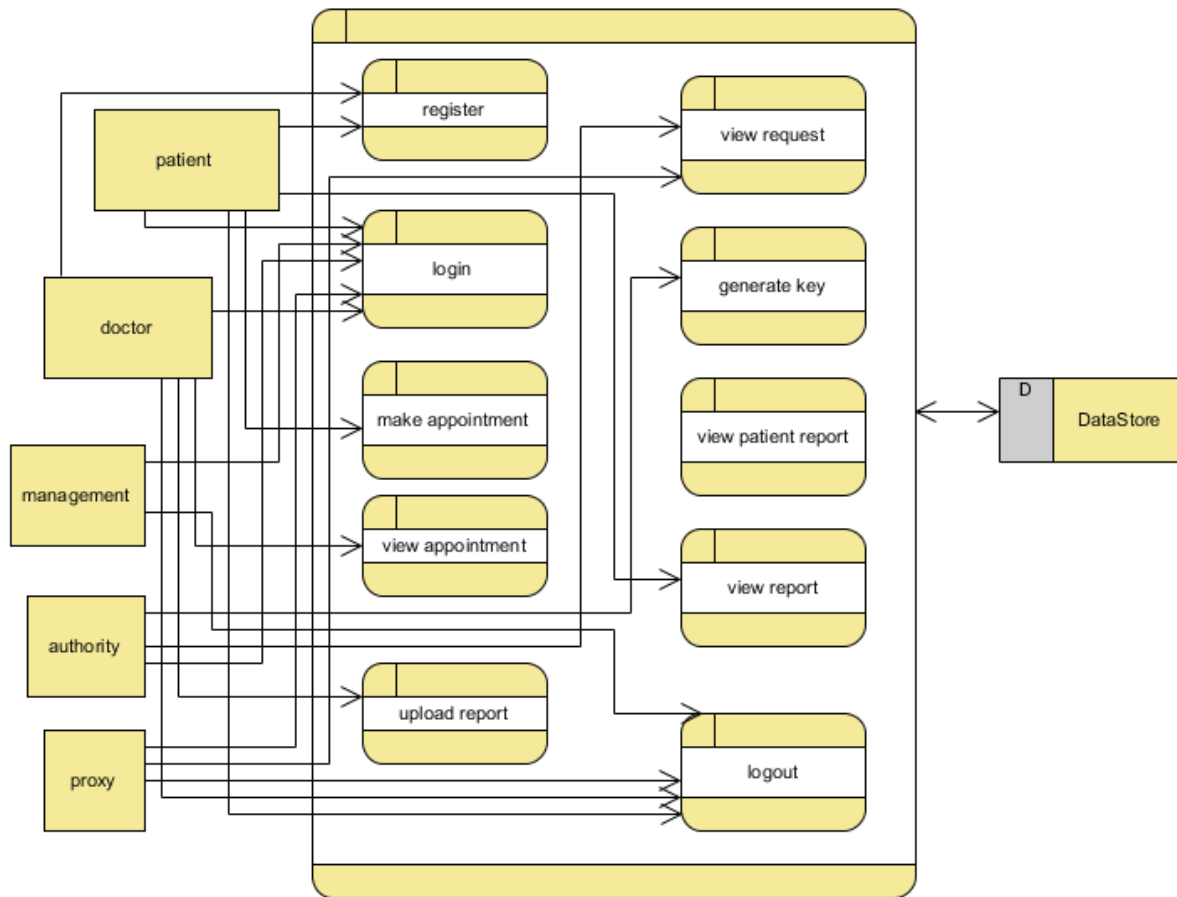
A Data Flow Diagram (DFD) is a traditional way to visualize the information flows within a system. A neat and clear DFD can depict a good amount of the system requirements graphically. It can be manual, automated, or a combination of both. It shows how information enters and leaves the system, what changes the information and where information is stored. The purpose of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communications tool between a systems analyst and any person who plays a part in the system that acts as the starting point for redesigning a system.

Level 1 Diagram:



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

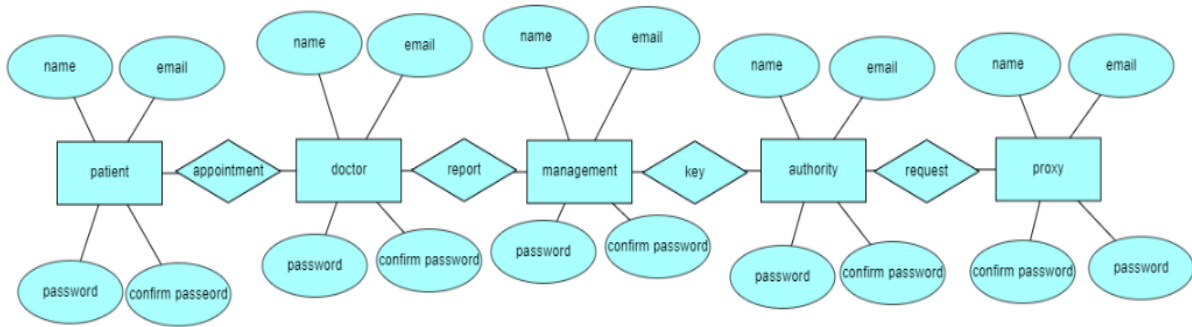
Level 2 Diagram:



b. ER diagram

An Entity–relationship model (ER model) describes the structure of a database with the help of a diagram, which is known as Entity Relationship Diagram (ER Diagram). An ER model is a design or blueprint of a database that can later be implemented as a database. The main components of E-R model are: entity set and relationship set.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING



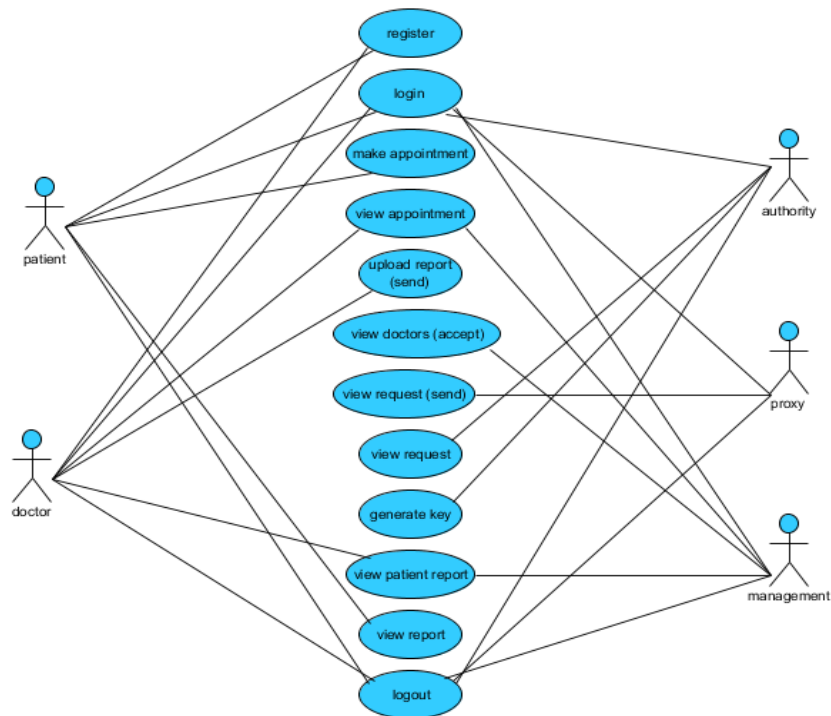
c. UML

- Uml stands for unified modeling language. Uml is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the object management group.
- The goal is for uml to become a common language for creating models of object oriented computer software. In its current form uml is comprised of two major components: a meta-model and a notation. In the future, some form of method or process may also be added to; or associated with, uml.
- The unified modeling language is a standard language for specifying, visualization, constructing and documenting the artifacts of software system, as well as for business modeling and other non-software systems.
- The uml represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

Use case diagram:

A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases. The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.

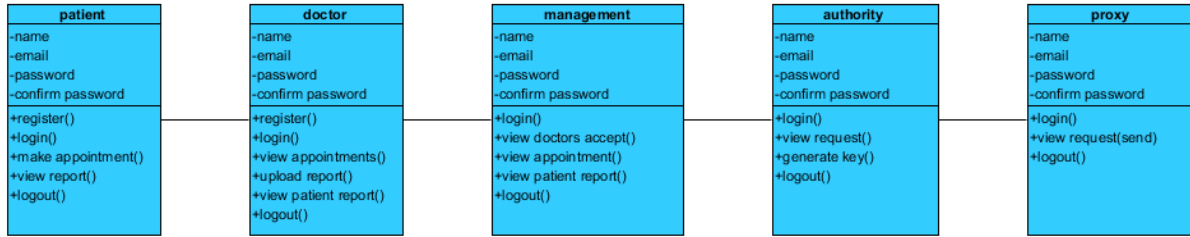
A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING



Class diagram:

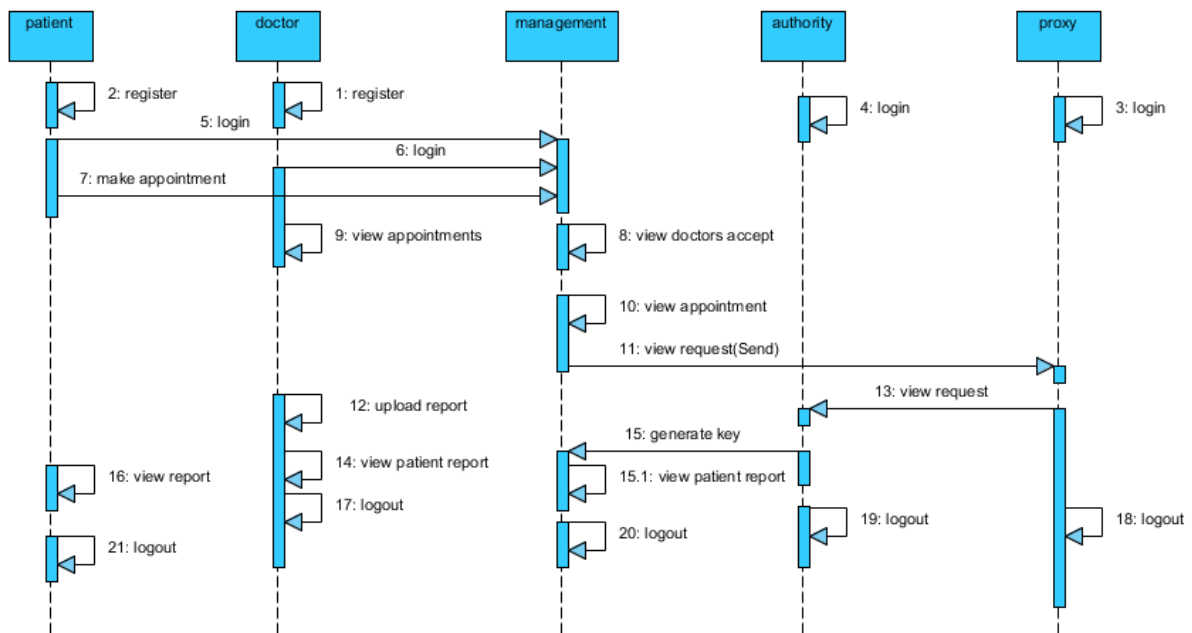
In software engineering, a class diagram in the unified modeling language (uml) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes. It explains which class contains information.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING



Sequence diagram:

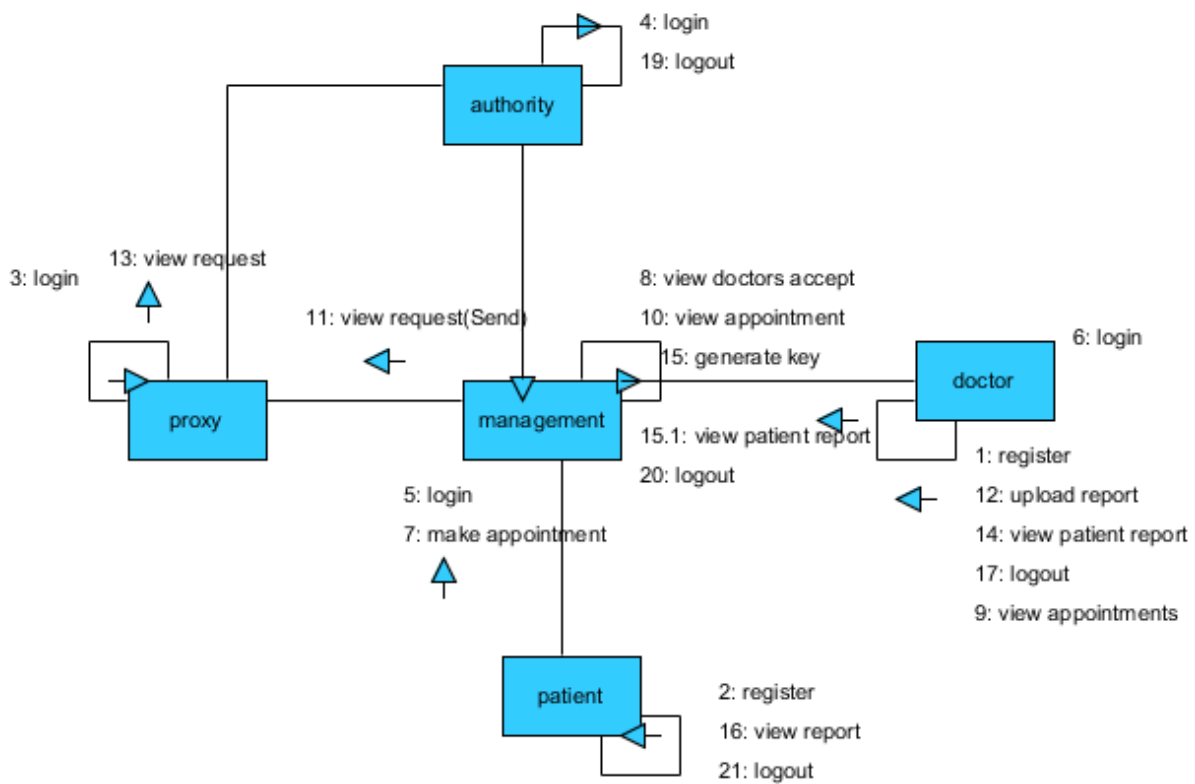
A sequence diagram in unified modeling language (uml) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a message sequence chart. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Collaboration diagram:

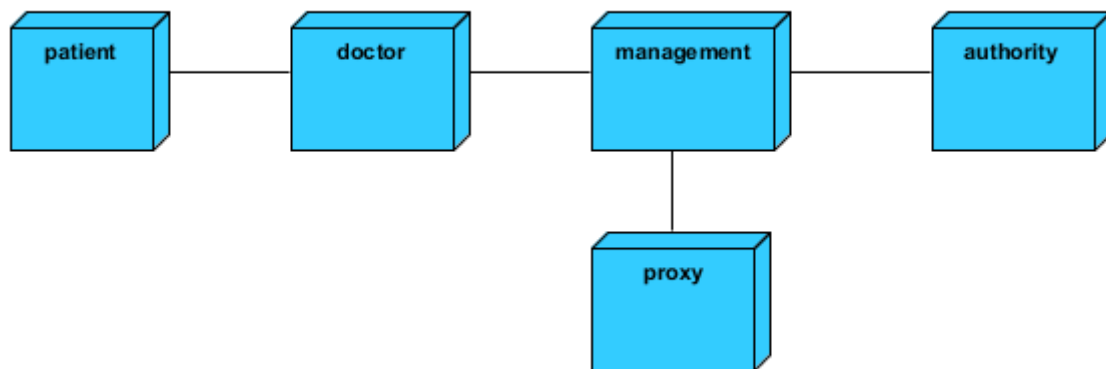
In collaboration diagram the method call sequence is indicated by some numbering technique as shown below. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram. The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization whereas the collaboration diagram shows the object organization.



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Deployment diagram:

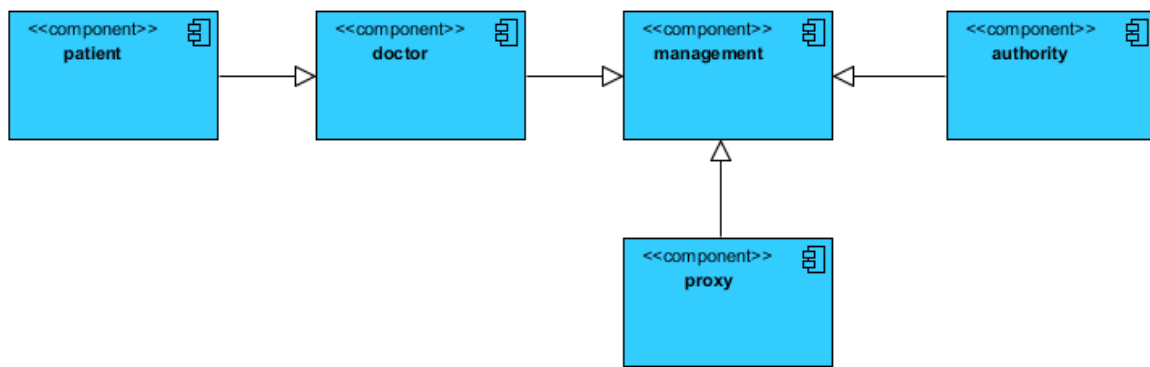
Deployment diagram represents the deployment view of a system. It is related to the component diagram. Because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware's used to deploy the application.



Component diagram:

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executable, libraries etc. So the purpose of this diagram is different, component diagrams are used during the implementation phase of an application. But it is prepared well in advance to visualize the implementation details. Initially the system is designed using different uml diagrams and then when the artifacts are ready component diagrams are used to get an idea of the implementation.

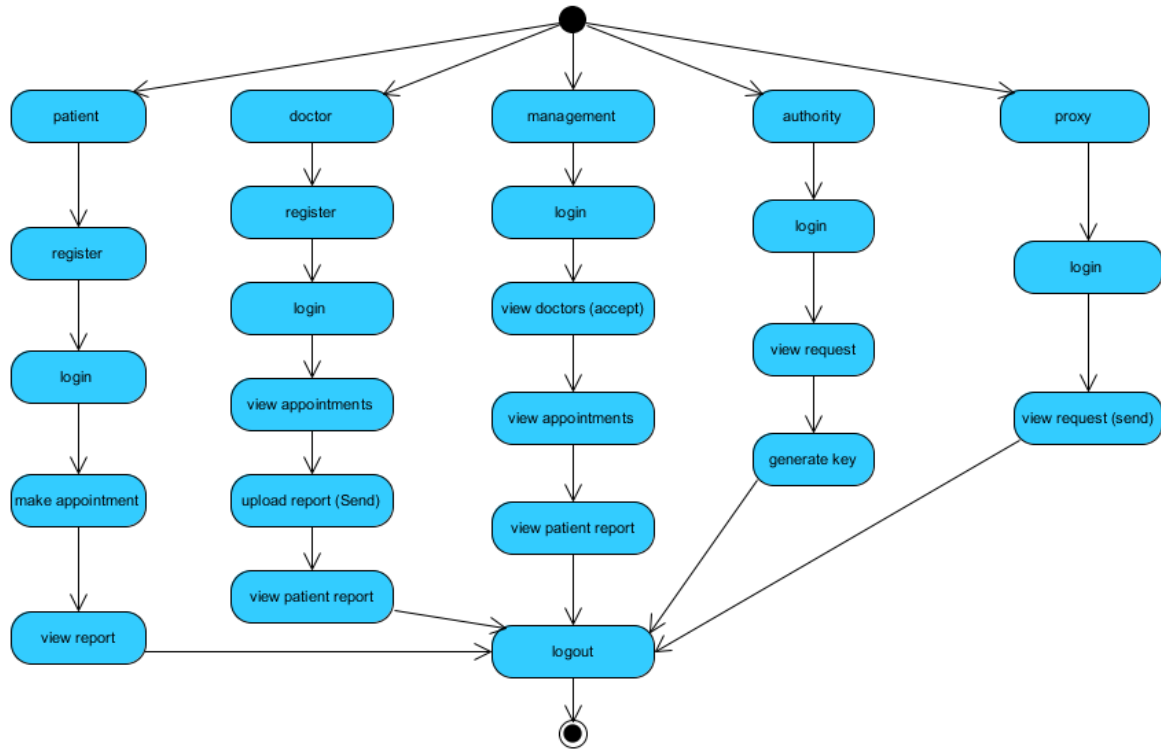
A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING



Activity diagram:

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the unified modeling language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING



• Data Dictionary

The **Data Dictionary** provides a detailed description of each database table and its corresponding fields. It defines data types, purposes, and constraints used in the system's backend for managing users, encrypted file uploads, and file access requests.

1. Appointments:

Field Name	Data Type	Description
id	int(200)	Unique identifier for each appointment (PK)
serialnumber	varchar(100)	Serial number assigned to the appointment
patientname	varchar(100)	Name of the patient

**A LIGHTWEIGHT POLICY UPDATE SCHEME FOR
OUTSOURCED PERSONAL HEALTH RECORDS SHARING**

Field Name	Data Type	Description
aadhar	varchar(100)	Aadhar number of the patient
bp	varchar(100)	Blood pressure reading
sugar	varchar(100)	Sugar level reading
hypertention	varchar(100)	Hypertension status (yes/no/level)
status1	varchar(100)	Appointment status (default: pending)
status2	varchar(100)	Follow-up or doctor status (default: pending)
billnumber	varchar(100)	Billing reference number

2. Connectdata:

Field Name	Data Type	Description
Id	int(100)	Unique ID (PK)
PatientName	varchar(100)	Name of the patient
PatientAge	varchar(100)	Age of the patient
Type	varchar(100)	Type of connection or request (e.g., follow-up)
status	varchar(100)	Current status (default: pending)

**A LIGHTWEIGHT POLICY UPDATE SCHEME FOR
OUTSOURCED PERSONAL HEALTH RECORDS SHARING**

3. Docreg

Field Name	Data Type	Description
slno	int(100)	Serial number/Primary key (PK)
Name	varchar(100)	Name of the doctor
Department	varchar(100)	Department or specialization
Age	varchar(100)	Age of the doctor
Number	varchar(100)	Contact number
Email	varchar(100)	Email address
Password	varchar(100)	Encrypted password
status	varchar(100)	Registration approval status (default: pending)

4. patient_reg:

Field Name	Data Type	Description
ID	int(200)	Unique patient ID (PK)
name	varchar(100)	Name of the patient
profile	varchar(100)	Path to profile image
profilename	varchar(100)	Filename of profile image
address	varchar(100)	Residential address

**A LIGHTWEIGHT POLICY UPDATE SCHEME FOR
OUTSOURCED PERSONAL HEALTH RECORDS SHARING**

Field Name	Data Type	Description
aadhar	varchar(100)	Aadhar number of the patient
bp	varchar(100)	Blood pressure status
sugar	varchar(100)	Sugar level
hypertention	varchar(100)	Hypertension condition
password	varchar(100)	Encrypted password

5. patientreq

Field Name	Data Type	Description
Id	int(200)	Request ID (PK)
Name	varchar(100)	Name of the patient
Type	varchar(100)	Type of request (e.g., consultation, follow-up)
Age	varchar(100)	Age of the patient
symptoms	varchar(100)	Reported symptoms
AppointmentDate	varchar(100)	Requested appointment date
Time	varchar(100)	Requested time slot
Status	varchar(100)	Request status (default: pending)

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

6. reports:

Field Name	Data Type	Description
Id	int(200)	Report ID (PK)
FileName	varchar(200)	Name of the uploaded report
FileData	longblob	Binary data of the report file
aadhar	varchar(200)	Patient's Aadhar number for reference
Status	varchar(200)	Status of the report (e.g., verified, pending)
Key1	varchar(100)	Associated encryption or verification key
Patientid	varchar(100)	Link to patient's ID

CHAPTER 6-TECHNOLOGY DESCRIPTION

The proposed project utilizes a set of modern and reliable technologies to ensure the secure management of outsourced Personal Health Records (PHRs) with efficient access control and policy updates. The selection of tools, languages, and platforms is made to provide scalability, ease of development, and data security in a cloud-based healthcare environment.

- **Programming Language:** The core development is done using **Python**, which provides robust support for backend logic, cryptographic operations, and modular development. Its extensive library support allows seamless implementation of functionalities such as encryption, file handling, and email integration.
 - **Technology Used:** Python 3.6+
- **Backend Framework:** The backend is developed using the **Django** web framework, which follows the MTV (Model-Template-View) architecture. Django simplifies the handling of routing, authentication, and ORM (Object-Relational Mapping), ensuring fast and secure development.
 - **Libraries Used:** Django, Pandas, os, smtplib
- **Frontend Technologies:** The user interface is designed using a combination of **HTML**, **CSS**, **Bootstrap**, and **JavaScript**. These technologies provide a responsive and visually clean interface for patients and doctors to interact with the system.
 - **Purpose:** Display forms, manage UI interactions, and render dynamic data.
- **Database:** The system uses **SQLite** as the database engine due to its simplicity and integration with Django. It stores user data, encrypted PHR metadata, doctor/patient records, and policy versioning history.
 - **Technology Used:** SQLite (lightweight and file-based)
- **Server Deployment:** For local testing and deployment, **XAMPP Server** is used to simulate the environment. This supports integration and testing of frontend-backend communication on a local machine.
 - **Role:** Hosting application locally for simulation

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

- **IDE/Development Tools:** The development is carried out using **PyCharm**, a powerful IDE that supports Python and Django development with features like intelligent code completion, version control, and debugging tools.
 - **Tool Used:** JetBrains PyCharm
- **Operating System Compatibility:** The application is compatible with commonly used Windows operating systems including **Windows 7, 8, and 10**, making it easy to deploy and use across standard desktop environments.

CHAPTER 7 – TESTING & DEBUGGING TECHNIQUES

Testing & Debugging Techniques

Testing and debugging play a vital role in ensuring the accuracy, reliability, and security of secure access control systems. In this project, the proposed CP-ABE and Proxy Re-Encryption based framework for Personal Health Records (PHRs) is thoroughly tested using a combination of functional, integration, and access-control validation techniques. Debugging is carried out throughout the development cycle to detect logic errors, flow mismatches, and ensure security behavior aligns with access policies.

Unit Testing

Each core module and functionality is tested in isolation to verify correct behavior. This includes:

- Encryption and decryption operations using CP-ABE.
- Access policy enforcement and attribute matching.
- Policy versioning and rollback functionalities.
- Re-encryption flow handling via the proxy server.

Integration Testing

After successful unit testing, modules are integrated and tested together to validate seamless end-to-end interactions.

Focus:

- Data encryption combined with user attribute verification.
- Proxy re-encryption during policy updates.
- Frontend-backend communication across Django views, templates, and database models.

Access Control & Policy Simulation Testing

The access control mechanism is validated by simulating real-world scenarios including:

- Authorized and unauthorized user attempts based on defined attribute policies.
- Policy change propagation and access behavior post re-encryption.
- Version rollback to previous policies and ensuring correct access restoration.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Expected Outcome: Only users matching the current policy attributes can decrypt and access the PHR, while others are restricted.

Debugging Techniques

- **Step-by-Step Execution:** Using PyCharm's debugger to trace variable values, monitor view logic, and evaluate backend responses.
- **Assertion Checks:** Applied during encryption/decryption and policy checks to ensure values match expected conditions.
- **Logging:** Implemented across sensitive processes such as file encryption, policy updates, and user authorization to monitor system behavior and trace bugs.

Test Case Design

- Test cases were created using different user roles (doctor, patient), access attributes (e.g., department), and policy versions.
- Negative test cases included unauthorized access, invalid file uploads, and incorrect decryption keys.
- Edge case testing ensured the system handled unexpected inputs and rejected unauthorized access reliably.

Performance Monitoring

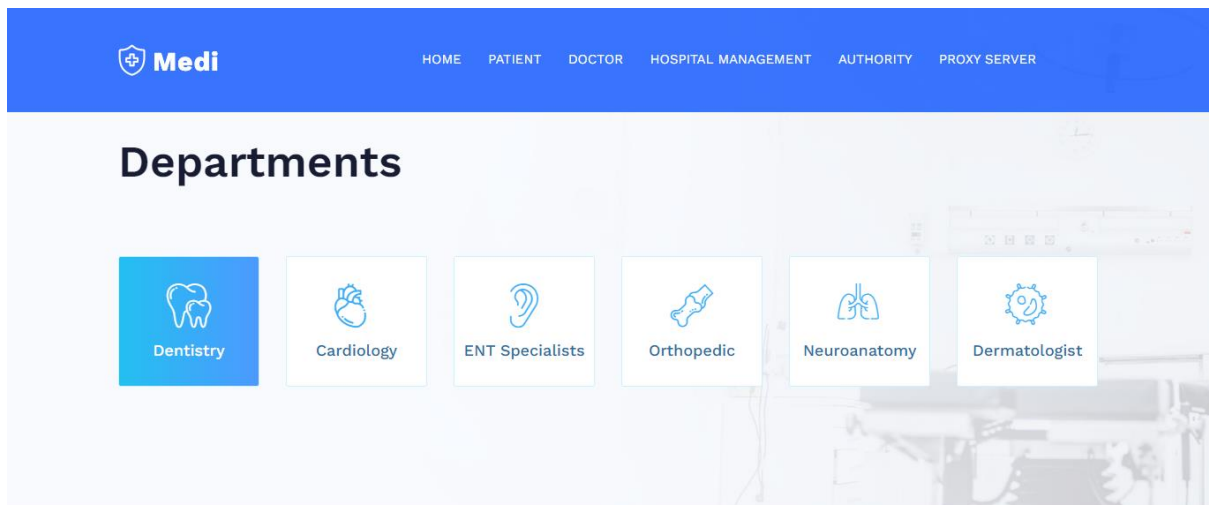
Although not the primary focus, encryption, decryption, and re-encryption times were observed during testing to ensure acceptable processing speeds. This ensures that even under policy updates, the system remains lightweight and responsive.

CHAPTER 8 – OUTPUT SCREENS

Home Page: Home page of the lightweight policy update scheme for outsourced personal health record sharing.

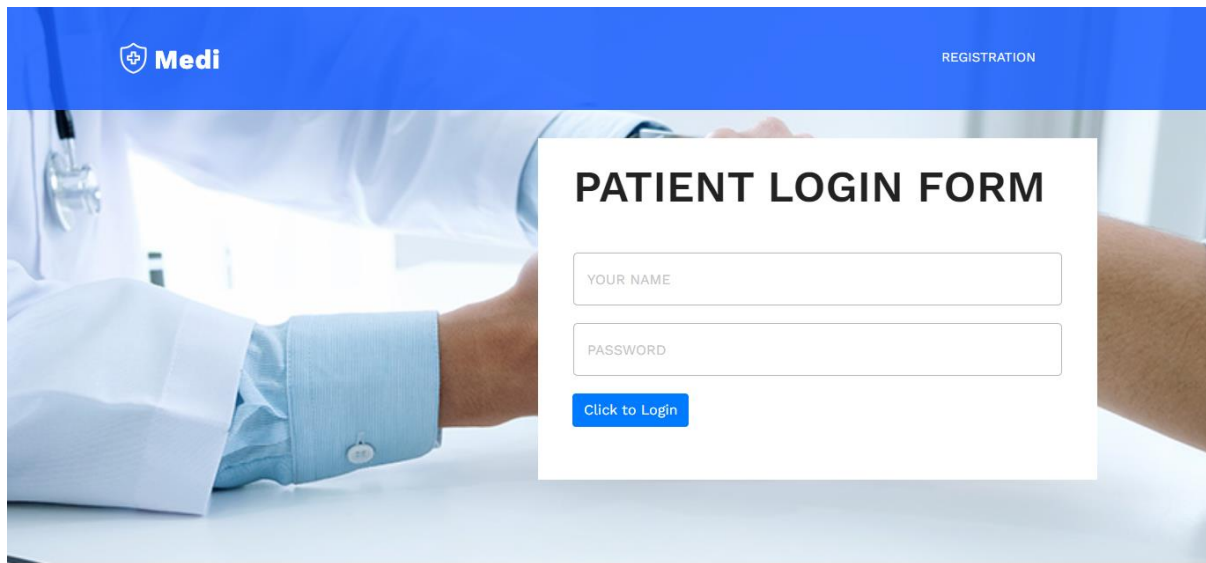


Modules Page: In home page user can see departments held in hospitals.



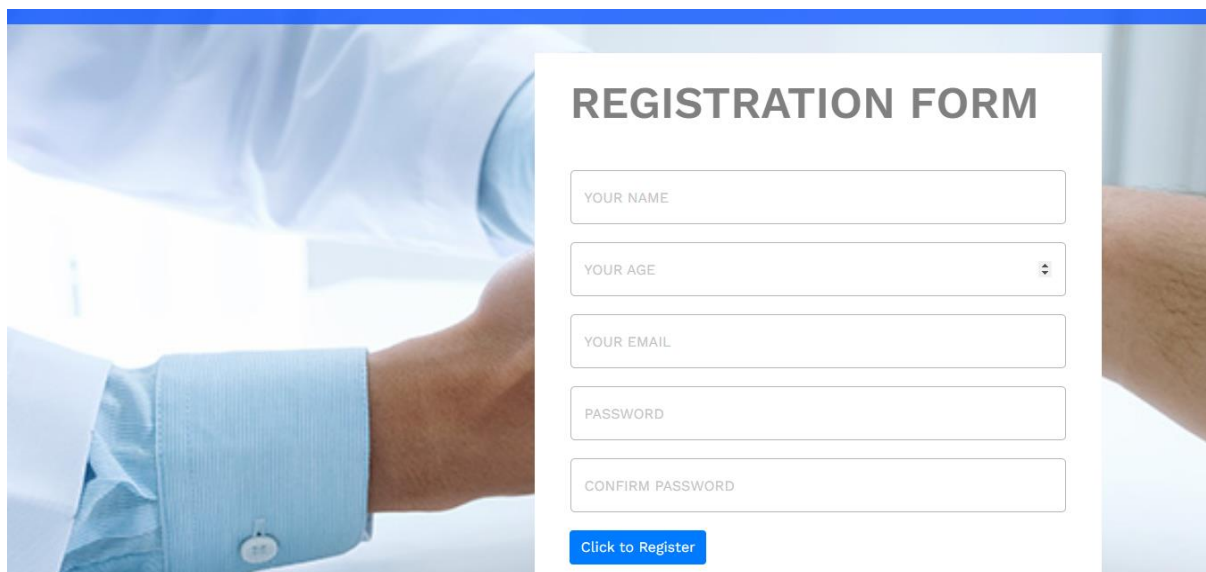
A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Patient Login Form: If patient have account, he can login with their valid credentials. Otherwise he can registered.



The image shows a web interface for a patient login form. At the top, there is a blue header bar with a white shield icon and the text "Medi" on the left, and the word "REGISTRATION" on the right. Below the header, the background is a blurred image of a doctor's arm in a white coat. Overlaid on this is a white rectangular box titled "PATIENT LOGIN FORM". Inside the box, there are two input fields: "YOUR NAME" and "PASSWORD". Below these fields is a blue button with the text "Click to Login".

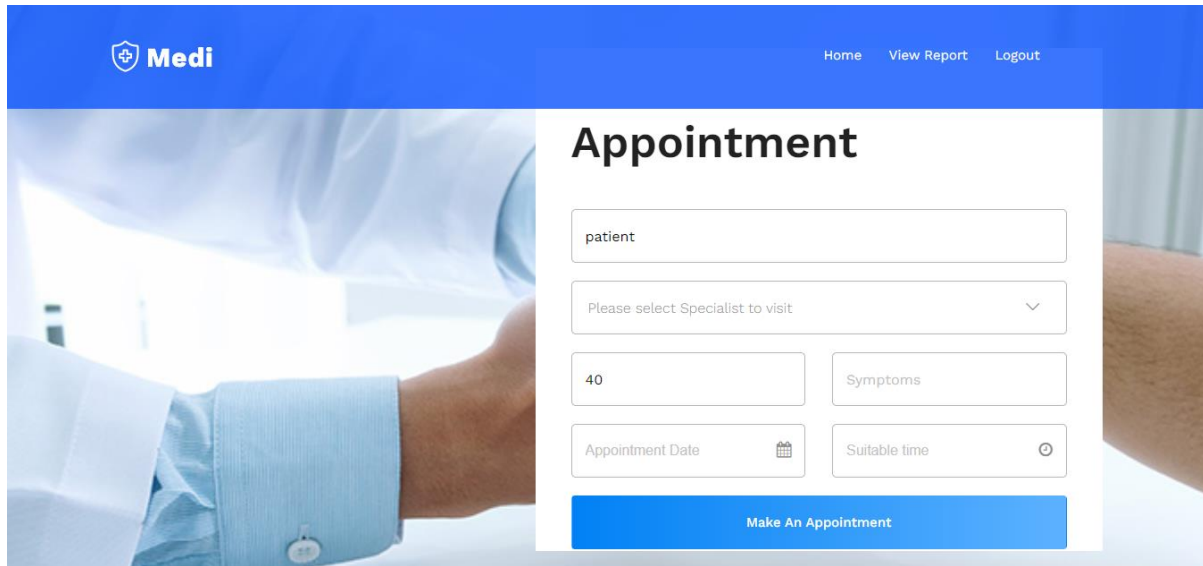
Patient Registration Page: If patient has no account, he can registered by giving name, age, email and password.



The image shows a web interface for a patient registration form. At the top, there is a blue header bar. Below the header, the background is a blurred image of a doctor's arm in a white coat. Overlaid on this is a white rectangular box titled "REGISTRATION FORM". Inside the box, there are five input fields: "YOUR NAME", "YOUR AGE" (with a dropdown arrow), "YOUR EMAIL", "PASSWORD", and "CONFIRM PASSWORD". Below these fields is a blue button with the text "Click to Register".

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Appointment Form: After login with their valid login credentials, he can make an appointment with doctor. Here patient can give their symptoms.



The screenshot shows the 'Appointment' form on the Medi website. The form is overlaid on a background image of a doctor's arm in a white coat. The form includes a header with the Medi logo and navigation links (Home, View Report, Logout). The form fields are: a text input for 'patient' (containing 'patient'), a dropdown menu for 'Please select Specialist to visit', a text input for '40', a text input for 'Symptoms', a date picker for 'Appointment Date', and a time picker for 'Suitable time'. A blue button at the bottom says 'Make An Appointment'.

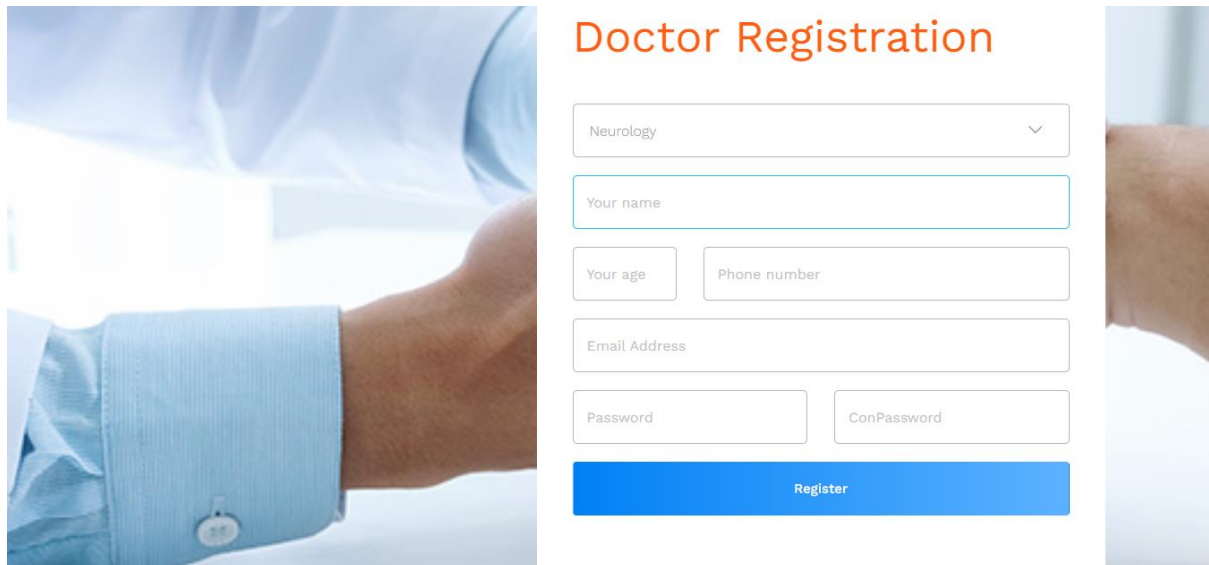
Doctor Login: After creating an account, doctor can login with valid login credentials. Here doctor can give their login details.



The screenshot shows the 'Doctor Login Form' on the Medi website. The form is overlaid on a background image of a doctor's arm in a white coat. The form includes a header with the Medi logo and navigation links (Home, Register). The form fields are: a text input for 'Your name', a text input for 'Password', and a blue button labeled 'Login'.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Doctor Registration Form: If doctor don't have an account, he can registered by giving their details.

The image shows a 'Doctor Registration' form overlaid on a background of a doctor's arm in a light blue shirt. The form has a title 'Doctor Registration' in orange. It includes a dropdown menu for 'Neurology', a text input for 'Your name', a 'Your age' input, a 'Phone number' input, an 'Email Address' input, a 'Password' input, and a 'ConPassword' input. A blue 'Register' button is at the bottom.

Doctor Registration

Neurology

Your name

Your age

Phone number

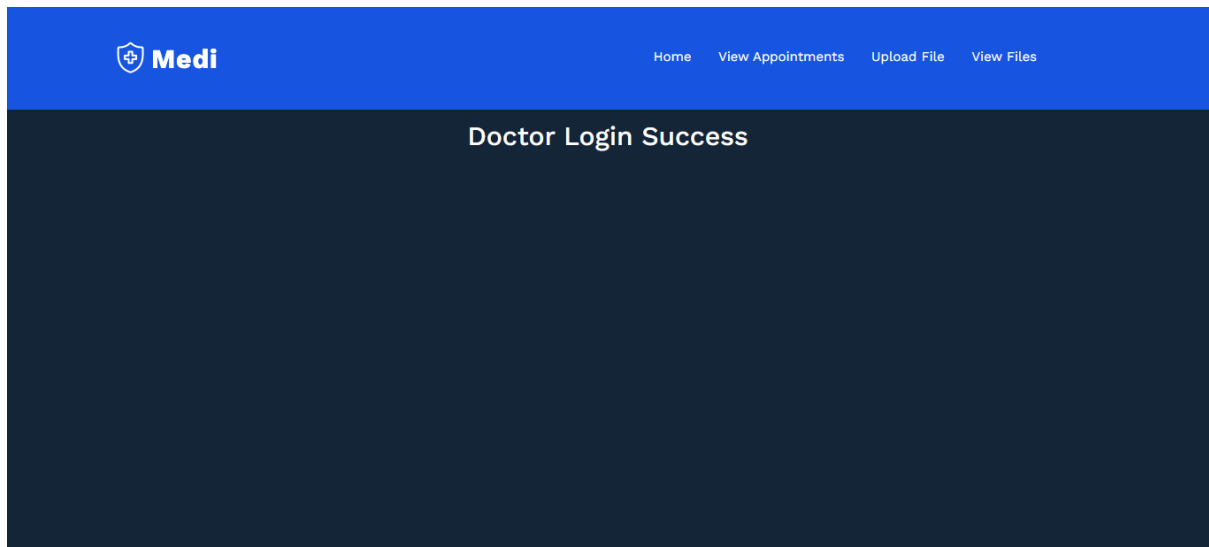
Email Address

Password

ConPassword

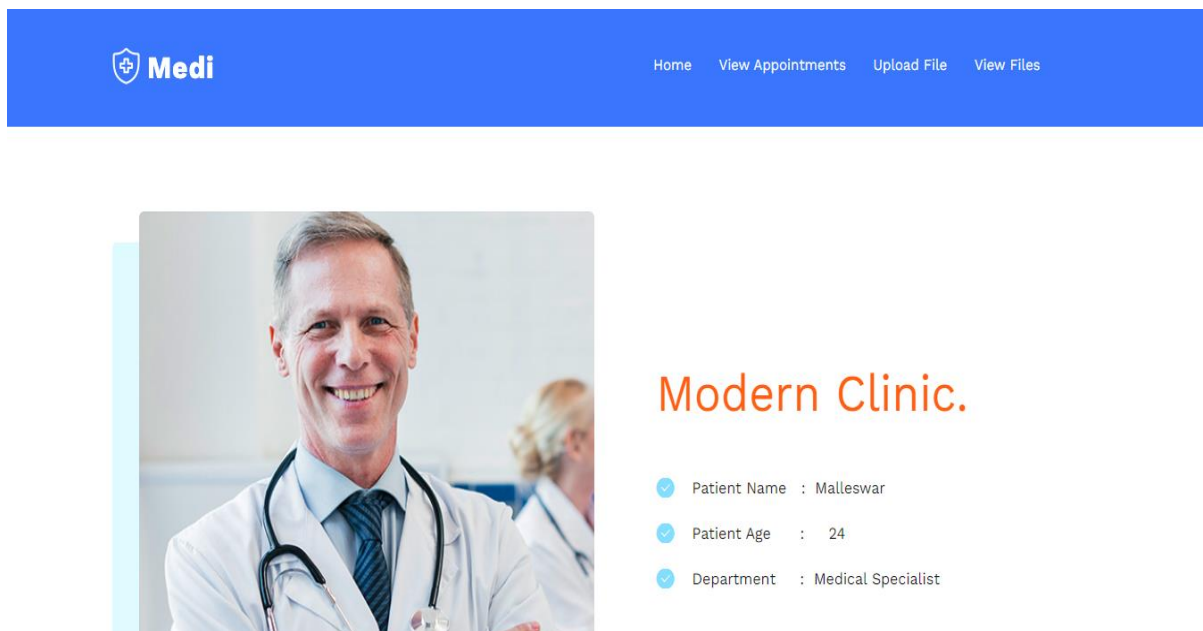
Register

Doc Home Page: Doctor after login with their valid credentials, he redirect to doctor home page.

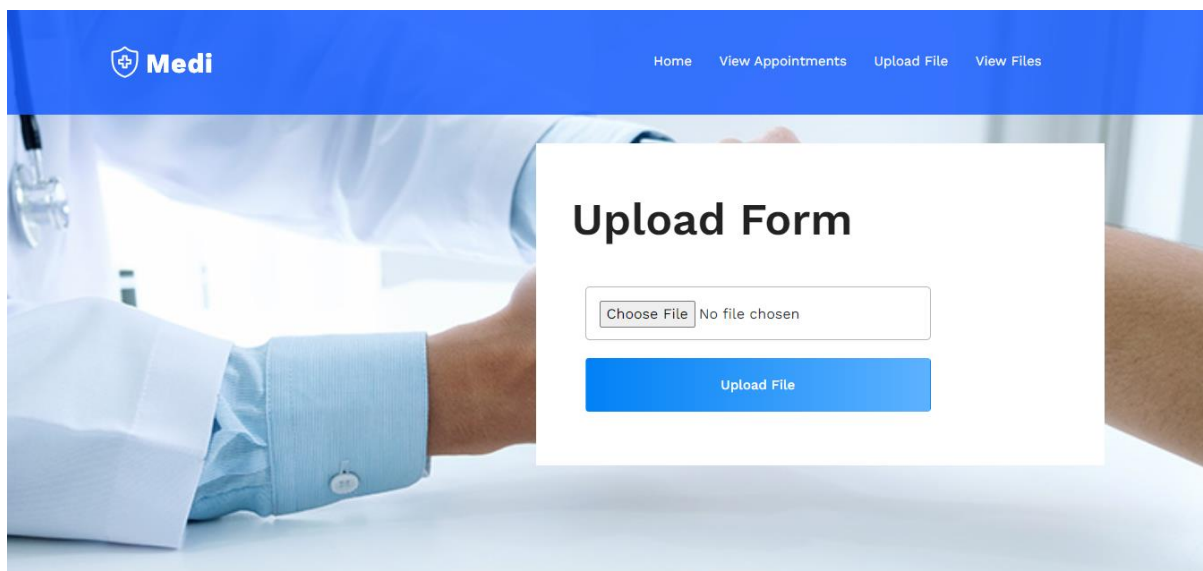


A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

View Appointments: Here doctor can view patient appointments.



Upload Files: After seeing patient symptoms, doctor can upload their reports.



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

View Files: Here doctor can view their uploaded reports, perform action on upload document.

The screenshot shows the 'View Files' page of the Medi application. The header is blue with the Medi logo and navigation links: Home, View Appointments, Upload File, and View Files. The main content area has a title 'Patient Information' in orange. Below it is a table with four columns: FileName, Filedata, PatientEmail, and Action. The table contains one row with the following data: FileName: document.txt, Filedata: b'\xa4\xf3^07\x84\x12\xa3* IFNK\xbc\xe9', PatientEmail: malleswar@gmail.com, and Action: Perform (in red). The background of the page is a blurred image of medical equipment.

FileName	Filedata	PatientEmail	Action
document.txt	b'\xa4\xf3^07\x84\x12\xa3* IFNK\xbc\xe9'	malleswar@gmail.com	Perform

View Report: Here user can download their report with help of key, this key send through mail at the time doctor upload report.

The screenshot shows the 'View Report' page of the Medi application. The header is blue with the Medi logo and navigation links: Home, View Report, and Logout. The main content area has a title 'Your Reports' in bold. Below it is a text box containing the message 'You Got Blood Cancer positive'. At the bottom of the text box is a 'Download' button.

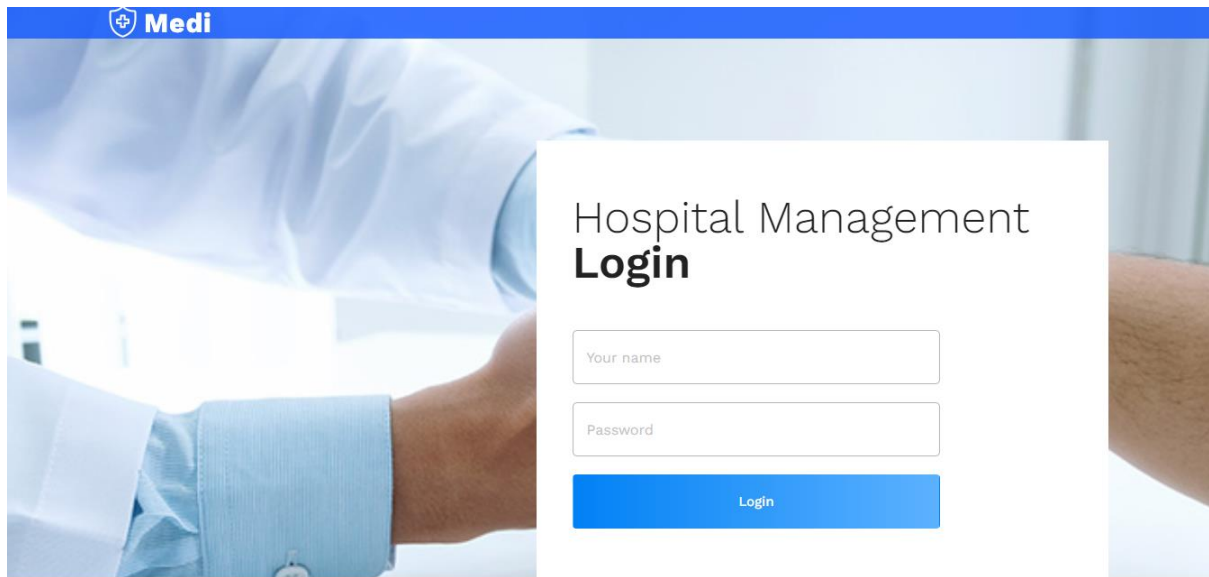
Your Reports

You Got Blood Cancer positive

Download

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Management Login: Here management login with their credentials, management have default login credentials.



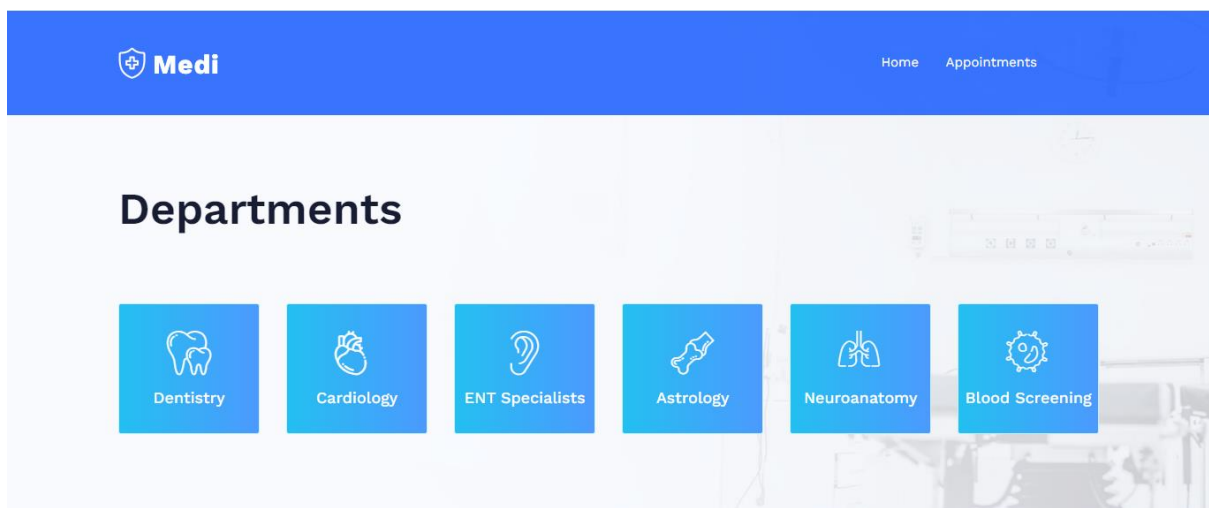
Hospital Management Login

Your name

Password

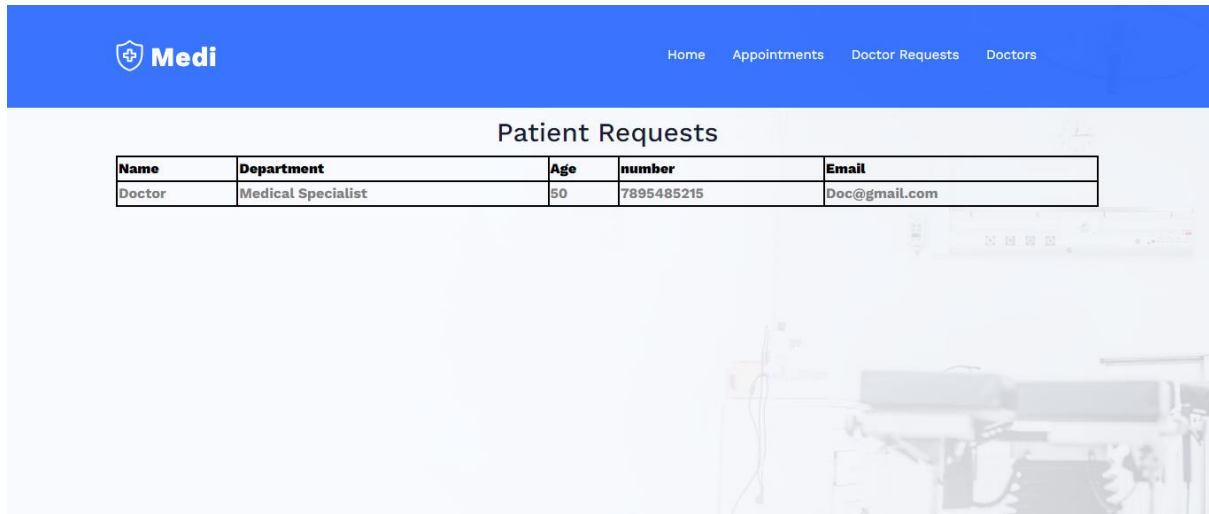
Login

Management Home Page: After management login with their login credentials successfully, he can redirect the management home page.



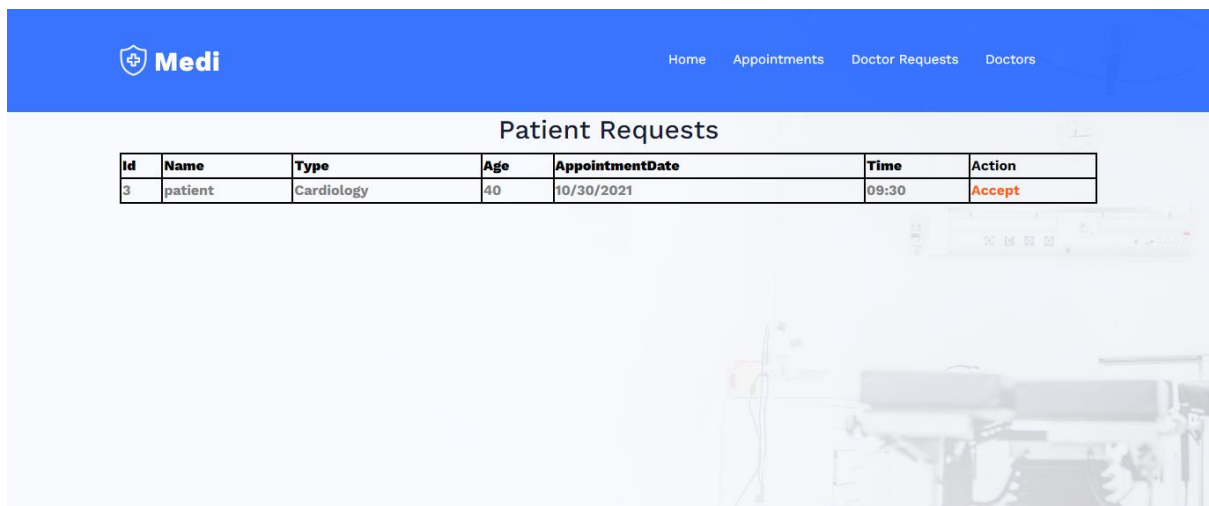
A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Patient Request: Patient make an appointment, it will show in management patient request. Here management accepts the request, After it can show the doctors.



The screenshot shows the Medi application interface. The header is blue with the Medi logo and navigation links: Home, Appointments, Doctor Requests, and Doctors. The main content area is titled 'Patient Requests' and displays a table with the following data:

Name	Department	Age	number	Email
Doctor	Medical Specialist	50	7895485215	Doc@gmail.com

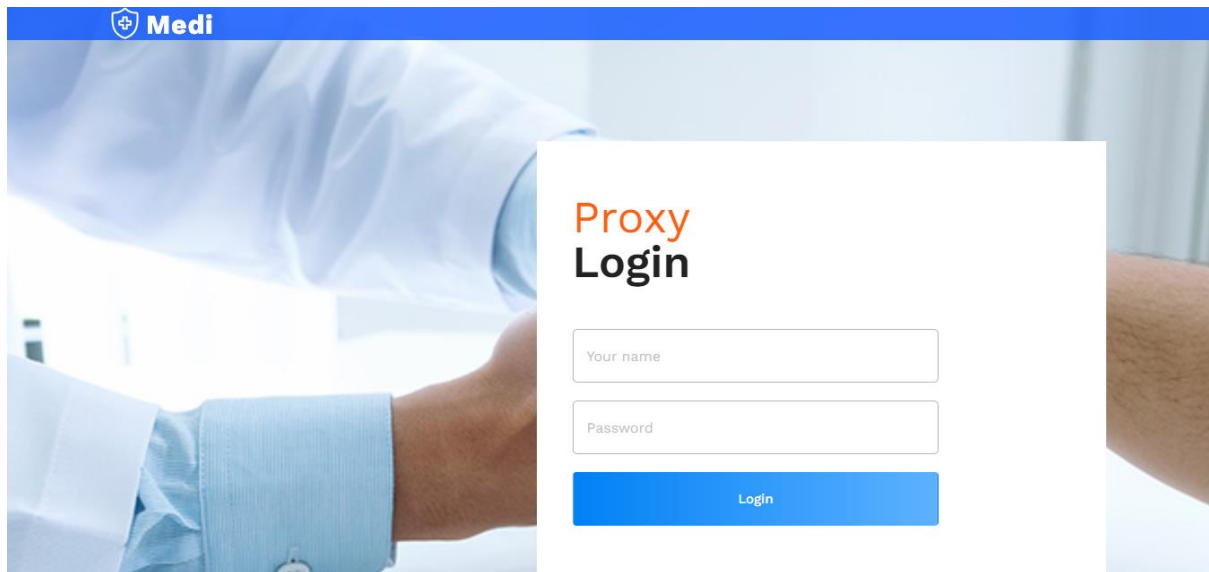


The screenshot shows the Medi application interface. The header is blue with the Medi logo and navigation links: Home, Appointments, Doctor Requests, and Doctors. The main content area is titled 'Patient Requests' and displays a table with the following data:

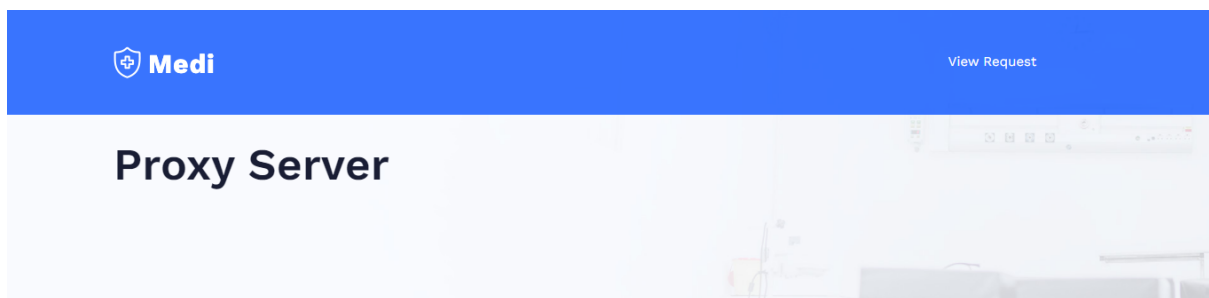
Id	Name	Type	Age	AppointmentDate	Time	Action
3	patient	Cardiology	40	10/30/2021	09:30	Accept

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Proxy Login: Here proxy have default login credentials, he use that login credentials for login.

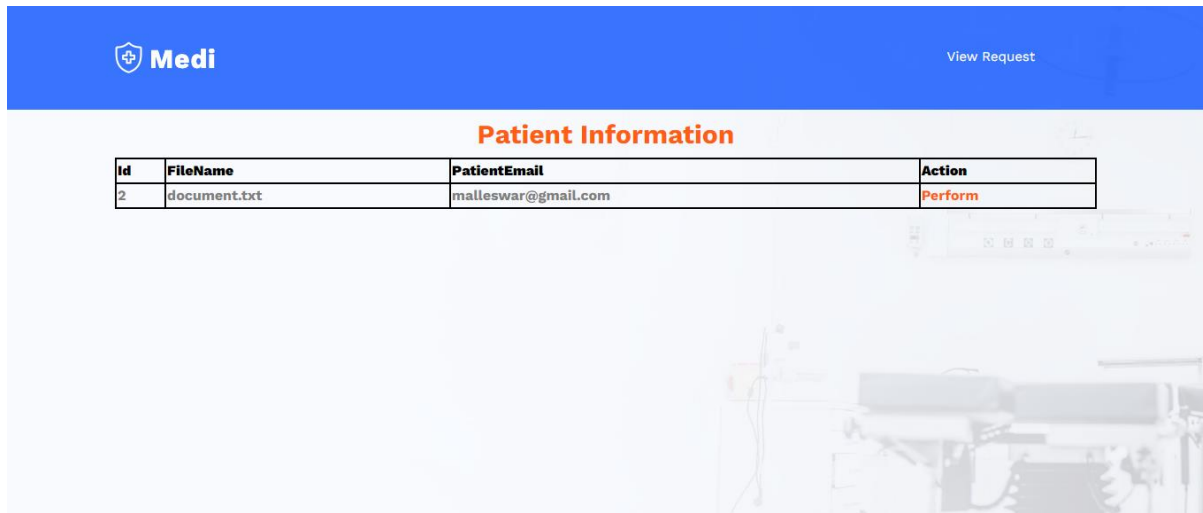
The image shows a web application interface for 'Medi'. At the top is a blue header with a white shield icon containing a cross and the text 'Medi'. Below the header is a large background image of a person's arm in a white lab coat. Overlaid on this is a white login box. The box has the title 'Proxy Login' in orange and black text. It contains two input fields: 'Your name' and 'Password'. Below these fields is a blue button with the text 'Login' in white.

View Request:

The image shows a web application interface for 'Medi'. At the top is a blue header with a white shield icon containing a cross and the text 'Medi'. On the right side of the header is a link that says 'View Request'. Below the header is a large background image of a medical server room. Overlaid on the left side of the background is the text 'Proxy Server' in a large, bold, black font.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

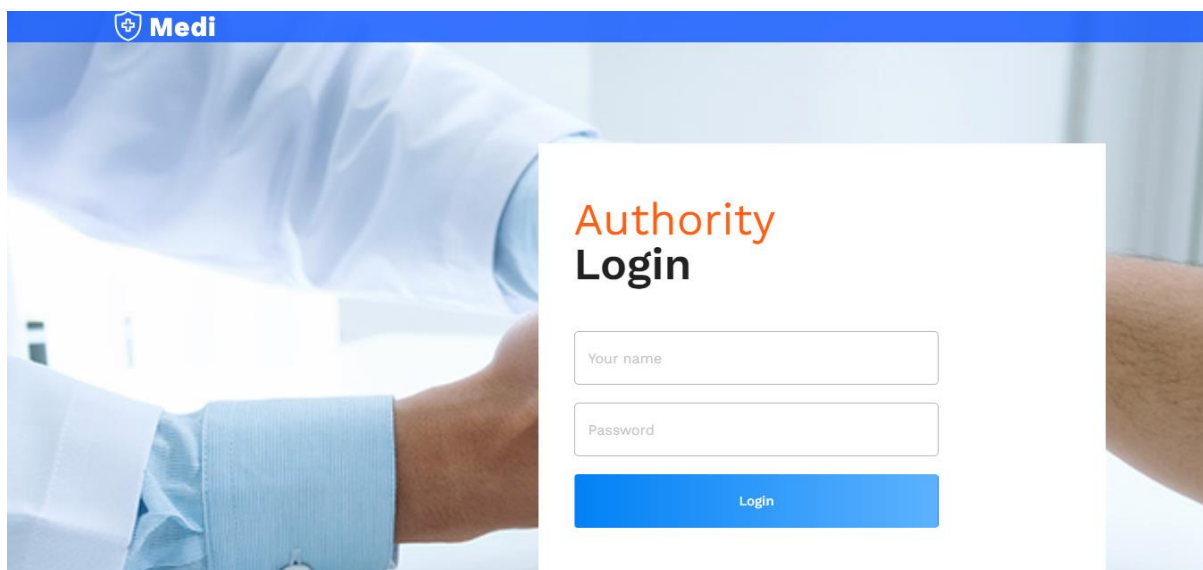
Requests:



The screenshot shows the 'Medi' interface with a blue header bar containing the 'Medi' logo and a 'View Request' link. Below the header, the title 'Patient Information' is displayed in orange. A table with four columns is shown: 'Id', 'FileName', 'PatientEmail', and 'Action'. The table contains one row of data. The 'Action' column for the first row contains a red 'Perform' link. The background of the interface is a blurred image of medical equipment.

Id	FileName	PatientEmail	Action
2	document.txt	malleswar@gmail.com	Perform

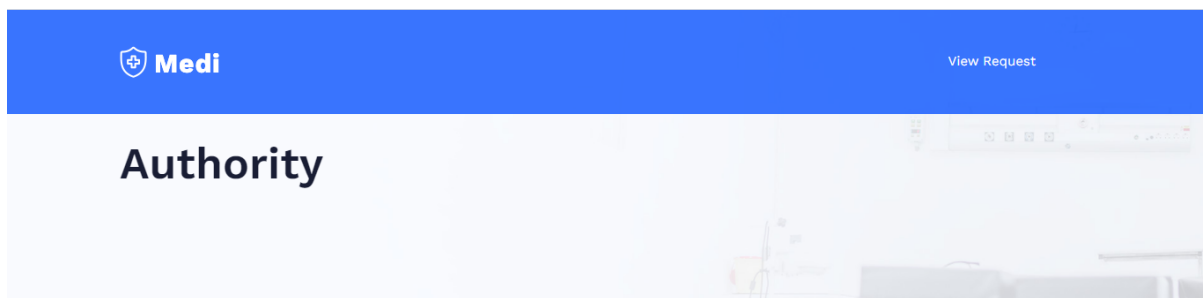
Authority Login: Here Authority have default login credentials, he use that login credentials for login.



The screenshot shows the 'Authority Login' form in the 'Medi' interface. The form is overlaid on a blurred background image of a person in a white lab coat. The form has a white background and contains the following elements: the title 'Authority Login' in orange and black text, a text input field labeled 'Your name', a text input field labeled 'Password', and a blue 'Login' button.

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Authority Home: After login, he can redirect to her home page.



Authority Request: Here authority can see requests, he can send that download key to patient through mail.



A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Logout:



CHAPTER 9 –CODE

```
import os
from flask import *
import mysql.connector
import pandas as pd
import random
from flask_mail import *

db = mysql.connector.connect(
    user='root', port=3306, database='lightweightpolicy')
cur = db.cursor()
app = Flask(__name__)
app.secret_key = '!@#H%S$BV#AS><)SH&BSGV*(_Sjnkxcb9+_ )84JSUHB&*%$^+= '

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/patient', methods=['POST', 'GET'])
def Patientlog():
    if request.method == 'POST':
        aadhar = request.form['aadhar']
        password = request.form['password']
        cur.execute(
            "select * from patient_reg where aadhar=%s and password=%s", (aadhar, password))
        content = cur.fetchall()
        db.commit()
        if content == []:
            msg = "Credentials Does't exist"
            return render_template('patientlog.html', msg=msg)
        else:
            print(content)
            aadhar = content[0][5]
            session['id'] = content[0][0]
            session['aadhar'] = aadhar
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
return render_template('patienthome.html', msg="Login success")
return render_template('patientlog.html')
```

```
@app.route('/patientreg', methods=['POST', 'GET'])
```

```
def Patientreg():
```

```
    if request.method == 'POST':
```

```
        name = request.form['name']
```

```
        profile = request.files['profile']
```

```
        profilename = profile.filename
```

```
        address = request.form['address']
```

```
        aadhar = request.form['aadhar']
```

```
        bp = request.form['bp']
```

```
        sugar = request.form['sugar']
```

```
        hypertention = request.form['hypertention']
```

```
        password = request.form['password']
```

```
        # profile.save('/static/uploadfiles/' + profile.filename)
```

```
        mypath = os.path.join("profiles/", profilename)
```

```
        profile.save(mypath)
```

```
        profilepath = "profiles/" + profilename
```

```
        sql = "select * from patient_reg where aadhar='%s' and password='%s'" % (
            aadhar, password)
```

```
        cur.execute(sql)
```

```
        data = cur.fetchall()
```

```
        db.commit()
```

```
        if data == []:
```

```
            sql = "insert into
```

```
patient_reg(name,profile,address,aadhar,bp,sugar,hypertention,profilename,password) values
(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)"
```

```
            val = (name, profilepath, address, aadhar, bp,
```

```
                sugar, hypertention, profilename, password)
```

```
            cur.execute(sql, val)
```

```
            db.commit()
```

```
            return render_template('patientlog.html')
```

```
        else:
```

```
            warning = 'Details already Exist'
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
return render_template('patientreg.html', msg=warning)
return render_template('patientreg.html')
```

```
@app.route("/myappointments", methods=["POST", "GET"])
def myappointments():
    if request.method == "POST":
        billnumber = request.form['billnumber']
        serialnumber = request.form['serialnumber']
        patientname = request.form['patientname']
        aadhar = request.form['aadhar']
        bp = request.form['bp']
        sugar = request.form['sugar']
        hypertention = request.form['hypertention']
        sql = "insert into
appointments(serialnumber,patientname,aadhar,bp,sugar,hypertention,billnumber)values(%s,
%s,%s,%s,%s,%s,%s,%s)"
        values = (serialnumber, patientname, aadhar,
            bp, sugar, hypertention, billnumber)
        cur.execute(sql, values)
        db.commit()
        sql = "select id,patientname,serialnumber,billnumber from appointments"
        cur.execute(sql)
        data = cur.fetchall()
        return render_template("myappointments.html", data=data)

sql = "select * from patient_reg"
cur.execute(sql)
data = cur.fetchall()
return render_template("myreport.html", data=data)
```

```
@app.route("/allreports")
def allreports():
    sql = "select Id,FileName,aadhar,status from reports"
    cur.execute(sql)
    data = cur.fetchall()
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
print(data)
return render_template("allreports.html", data=data)

@app.route("/download1/<int:aadhar>")
def download1(aadhar=0):
    sql = "select filename from reports where aadhar='%s'" % (aadhar)
    cur.execute(sql)
    filename = cur.fetchall()[0]

    filename = filename[0]
    # Assuming files are stored in a directory named 'files' in the same directory as your Flask
    application
    file_path = f"uploads/{filename}"
    try:
        with open(file_path, 'rb') as file:
            content = file.read()
            print(f"File content: {content}") # or log this information
    except Exception as e:
        print(f"Error reading file: {e}") # or log this information
    return send_file(file_path, as_attachment=True)

@app.route('/patientreq', methods=['POST', 'GET'])
def patientreq():
    if request.method == 'POST':
        Name = request.form['Name']
        doc = request.form['Doc']
        Age = request.form['Age']
        Symptoms = request.form['symptoms']
        AppointmentDate = request.form['AppointmentDate']
        Time = request.form['Time']
        sql = "insert into patientreq (Name,Type,Age,symptoms,AppointmentDate,Time) values
        ('%s','%s','%s','%s','%s','%s')" % (
            Name, doc, Age, Symptoms, AppointmentDate, Time)
        cur.execute(sql)
        db.commit()
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
msg = "Your appointment request Sent to Management"
return render_template('patienthome.html', msg=msg)
return render_template('patienthome.html')
```

```
@app.route('/labtechnician', methods=['POST', 'GET'])
def labtechnician():
    if request.method == 'POST':
        useremail = request.form['useremail']
        password = request.form['passcode']
        if useremail == "technician@gmail.com" and password == 'technician':
            return render_template('managementhome.html')
        return render_template('management.html')
```

```
@app.route("/patientreports")
def patientreports():
    sql = "select id,billnumber,serialnumber,patientname,aadhar,status1 from appointments"
    cur.execute(sql)
    data = cur.fetchall()
    return render_template("patientreports.html", data=data)
```

```
@app.route("/download/<int:aadhar>/")
def download(aadhar=0):
    print(aadhar)
    sql = "select filename from reports where aadhar='%s'" % (aadhar)
    cur.execute(sql)
    filename = cur.fetchall()[0]

    filename = filename[0]
    # Assuming files are stored in a directory named 'files' in the same directory as your Flask
    application
    file_path = f"uploads/{filename}"
    try:
        with open(file_path, 'rb') as file:
            content = file.read()
```


A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
        print(f'File content: {content}') # or log this information
    except Exception as e:
        print(f'Error reading file: {e}') # or log this information

    return send_file(file_path, as_attachment=True)

@app.route("/vreport/<x>")
def vreport(x=0):
    sql = "select Id,AES_DECRYPT(FileData, 'sec_key') from reports where Id='%s'" % (
        x)
    data = pd.read_sql_query(sql, db)
    return render_template("vreport.html", cols=data.columns.values,
        rows=data.values.tolist())

@app.route('/UploadReports')
def UploadReports():
    sql = "select id,patientname,serialnumber,billnumber,status1 from appointments"
    data = pd.read_sql_query(sql, db)
    db.commit()
    return render_template('viewappointments.html', cols=data.columns.values,
        rows=data.values.tolist())

@app.route('/accept_request/<x>/<y>/<z>')
def acceptreq(x=0, y="", z=""):
    print(x, y)
    session["rowid"] = x
    session['username'] = y
    print(session["rowid"])
    print(z)
    sql = "select Name,Department,Email from docreg where Department='%s' " % (
        z)
    data = pd.read_sql_query(sql, db)
    db.commit()
    print(data)
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
if data.empty:
    flash('Doctot is not available')
    return redirect(url_for('view_appointments'))
else:
    sql = "update patientreq set status='accepted' where status='pending' and Id='%s' and
Name='%s'" % (
        x, y)
    cur.execute(sql)
    db.commit()
    return render_template('acptreq.html', cols=data.columns.values, rows=data.values.tolist())

@app.route('/Connect/<x>/<y>/<z>')
def mergereq(x="", y="", z=""):
    print("=====")
    print(x)
    print(y)
    print(z)
    sql = "select name,Type,Age from patientreq where status='accepted' and Name='%s'" % (
        session['username'])
    cur.execute(sql)
    da = cur.fetchall()
    db.commit()
    dat = [j for i in da for j in i]
    print(dat)

    sql = "insert into connectdata(Patientname,patientAge,Type)values('%s','%s','%s')" % (
        dat[0], dat[2], dat[1])
    cur.execute(sql)
    db.commit()

    return redirect(url_for('view_appointments'))

@app.route('/Doc_requests')
def Docrequests():
    sql = "select Name,Department,Age,Number,Email from docreg where status='pending'"

```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
data = pd.read_sql_query(sql, db)
db.commit()
return render_template('Doc.html', cols=data.columns.values, rows=data.values.tolist())

@app.route('/acpt_doc/<x>/<y>')
def acceptdoc(x="", y=""):
    sql = "update docreg set status='accepted' where status='pending' and Name='%s' and
Email='%s'" % (
        x, y)
    cur.execute(sql)
    db.commit()
    sender_address = 'sender@gmail.com'
    sender_pass = 'password'
    content = "Your Request Is Accepted by the Management Plas You Can Login Now"
    receiver_address = y
    message = MIMEMultipart()
    message['From'] = sender_address
    message['To'] = receiver_address
    message['Subject'] = "A Lightweight Policy Update Scheme for Outsourced Personal
Health Records Sharing project started"
    # message.attach(MIMEText(content, 'plain'))
    # ss = smtplib.SMTP('smtp.gmail.com', 587)
    # ss.starttls()
    # ss.login(sender_address, sender_pass)
    # text = message.as_string()
    # ss.sendmail(sender_address, receiver_address, text)
    # ss.quit()
    return redirect(url_for('Docrequests'))

@app.route('/Docs')
def Docs():
    sql = "select Name,Department,Age,number,Email from docreg where status='accepted'"
    data = pd.read_sql_query(sql, db)
    db.commit()
    return render_template("docs.html", cols=data.columns.values, rows=data.values.tolist())
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
@app.route('/adminlog', methods=['POST', 'GET'])
def adminlog():
    if request.method == 'POST':
        adminemail = request.form['adminemail']
        adminpassword = request.form['adminpassword']
        if adminemail == "admin@gmail.com" and adminpassword == "admin":
            return render_template("docrequest.html")
        else:
            return render_template("doctorlog.html")
    return render_template('doctorlog.html')

@app.route("/dashboard")
def dashboard():
    sql = "select id,patientname,serialnumber,billnumber from appointments"
    cur.execute(sql)
    data = cur.fetchall()
    return render_template("dashboard.html", data=data)

@app.route("/patientreport", methods=["POST", "GET"])
def patientreport():
    if request.method == "POST":
        keyvalue = request.form['keyvalue']
        sql = "select Id,Filename,AES_DECRYPT(FileData, 'sec_key') from reports where
Patientid='%s'" % (
            keyvalue)
        # sql="select Filedata from reports where Patientid='%s'"%(keyvalue)
        data = pd.read_sql_query(sql, db)
        return render_template("patientreport.html", x=2, cols=data.columns.values,
rows=data.values.tolist())
    return render_template("patientreport.html", x=1)

@app.route('/view_patient')
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
def viewpatient():
    sql = "select * from connectdata where Type='%s' and status='pending'" % (
        session['dept'])
    cur.execute(sql)
    data = cur.fetchall()
    db.commit()
    print(data)
    if data == []:
        msg = "You dont have any appointments "
        return render_template("viewpatient.html", msg=msg)
    Name = data[0][1]
    Age = data[0][2]
    Type = data[0][3]
    return render_template('viewpatient.html', name=Name, age=Age, type=Type)

@app.route("/patient_access/<a>/<b>")
def patientaccess(a="", b=0):
    sql = "select Email from patient_reg where Name='%s' and Age='%s'" % (a, b)
    cur.execute(sql)
    data = cur.fetchall()
    db.commit()
    print(data)
    if data != []:
        Email = data[0][0]
        session['email'] = Email
        sql = "update connectdata set status='accept' where status='pending' and
PatientName='%s'" % (
            a)
        cur.execute(sql)
        db.commit()
        return render_template("uploadfile.html", email=Email)

    return render_template("uploadfile.html")

@app.route('/upload_file/<int:id>/')
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
def uploadfile(id=0):
    print(id)
    return render_template('uploadfile.html', id=id)

@app.route("/reportuploadfile", methods=["POST", "GET"])
def reportuploadfile():
    if request.method == 'POST':
        id = request.form['id']
        filedata = request.files['filedata']
        n = filedata.filename
        data = filedata.read()
        print("=====")
        print(data)
        path = os.path.join("uploads/", n)
        filedata.save(path)
        status = "accepted"
        id1 = random.randint(0000, 9999)
        PatientId = "PID" + str(id1)
        sql = "insert into reports(FileName,FileData,aadhar,Status,Patientid)
values(%s,AES_ENCRYPT(%s,'sec_key'),%s,%s,%s)"
        val = (n, data, session['aadhar'], status, PatientId)
        cur.execute(sql, val)
        db.commit()
        msg = "file Uploaded successfully"
        print(msg)
        sql = "update appointments set status1='accepted' where id='%s'" % (id)
        cur.execute(sql)
        db.commit()
        return redirect(url_for("UploadReports"))

@app.route('/view_files')
def viewfiles():
    sql = "select Id,FileName,Filedata,PatientEmail from reports where PatientEmail='%s' and
status='accepted'" % (
        session['email'])
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
data = pd.read_sql_query(sql, db)
db.commit()
return render_template('files.html', cols=data.columns.values, rows=data.values.tolist())
```

```
@app.route('/performs/<d>')
def performs(d=0):
    print(d)
    sql = "update reports set status='updated' where Id='%s' and PatientEmail='%s'" % (
        d, session['email'])
    cur.execute(sql)
    db.commit()
    return redirect(url_for('viewfiles'))
    # return render_template('performs.html')
```

```
@app.route('/authority', methods=['POST', 'GET'])
def authority():
    if request.method == 'POST':
        name = request.form['Username']
        password = request.form['passcode']
        if name == 'Authority' and password == 'auth':
            return render_template('authhome.html')

    return render_template('authority.html')
```

```
@app.route('/vr')
def vr():
    sql = "select Id,FileName,PatientEmail from reports where status='updated'"
    data = pd.read_sql_query(sql, db)
    db.commit()
    return render_template('vr.html', cols=data.columns.values, rows=data.values.tolist())
```

```
@app.route('/proxy_server', methods=['POST', 'GET'])
def proxyserver():
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
if request.method == 'POST':
    name = request.form['Username']
    password = request.form['passcode']
    if name == "proxy" and password == "server":
        return render_template('proxylog.html')
    return render_template('proxy.html')
```

```
@app.route('/Generate_Key/<c>/')
def generatekey(c=0):
    x = random.randrange(000000, 999999)
    print(x)
    print(c)
    sql = "update reports set Key1='%s',status='done' where Id = '%s' and status='updated' " %
    (
        x, c)
    cur.execute(sql)
    db.commit()

    return redirect(url_for('vr'))
```

```
@app.route('/all_requests')
def allrequests():
    sql = "select Id,FileName,PatientEmail,Key1 from reports where status='done' and
PatientEmail='%s'" % (
    session['email'])
    data = pd.read_sql_query(sql, db)
    db.commit()
    return render_template('all.html', cols=data.columns.values, rows=data.values.tolist())
```

```
@app.route('/sentmail/<e>/<k>/<z>')
def sentmail(e="", k=0, z=0):
    sender_address = 'balarampanigrahy42@gmail.com'
    sender_pass = 'Balaram@123'
    sql = "select PatientId from reports where Id='%s'" % (z)
```


A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
cur.execute(sql)
xyz = cur.fetchall()
db.commit()
print(xyz)
patientkey = xyz[0][0]
content = str(k) + "\n Your Key" + str(patientkey)
print(z)
print(content)
receiver_address = e
message = MIMEMultipart()
message['From'] = sender_address
message['To'] = receiver_address
message['Subject'] = "A Lightweight Policy Update Scheme for Outsourced Personal
Health Records Sharing project started"
# message.attach(MIMEText(content, 'plain'))
# ss = smtplib.SMTP('smtp.gmail.com', 587)
# ss.starttls()
# ss.login(sender_address, sender_pass)
# text = message.as_string()
# ss.sendmail(sender_address, receiver_address, text)
# ss.quit()
sql = "update reports set Status='complete' where Id='%s' and PatientEmail='%s'" % (
    z, session['email'])
cur.execute(sql)
db.commit()
return redirect(url_for("allrequests"))

@app.route('/myprofile', methods=['POST', 'GET'])
def myprofile():
    sql = "select * from patient_reg where aadhar='%s'" % (session['aadhar'])
    cur.execute(sql)
    data = cur.fetchall()
    db.commit()
    return render_template('report.html', data=data)
```

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

```
@app.route('/logout')
def logout():
    return redirect(url_for('index'))

if __name__ == "__main__":
    app.run(debug=True, port=8000)
```

CHAPTER 10 – CONCLUSION

Other than being a lightweight approach to policy updates for secure management of Personal Health Records in the cloud environment, this paper presents a new system combining CP-ABE and PRE for fine-grained access control and confidentiality for efficient management of access policies. Another major contribution of this work has been the offloading of the duty of the policy update and re-encryption to the proxy server, which significantly helps to reduce the computational overhead on the data owner. This enables efficient handling of the much larger datasets without laying the burden of resource-demanding encryption operations onto the patient or the healthcare provider. The system further interoperates with a policy versioning mechanism to ensure that all the effects of policy changes are traceable, thus giving an audit trail to access control updates. This capability strengthens the security of the system and provides accountability by allowing users to trace and revert old versions of policies. Experimental results have proven the ability of this system to accomplish the rather complicated task of securely sharing sensitive health data across multiple users while imposing low overhead with high scalability. It operates with an easy user interface, granting a smooth experience for the patient, doctor, and healthcare administrator, seamlessly helping them to manage and access health records securely. In conclusion, the scheme proposed in this paper provides a solution that is secure, efficient, and scalable, conversing the major concerns surrounding cloud-based health data management. It attenuates the complexities entailed by traditional encryption methods by providing a strong framework for the management and sharing of personal health records; as such, it is the solution of choice for modern healthcare systems.

CHAPTER 11 – BIBLIOGRAPHY

1. Bhatia, T., Verma, A. K., & Sharma, G. (2018). Secure sharing of mobile personal healthcare records using certificateless proxy re-encryption in cloud. *Transactions on Emerging Telecommunications Technologies*, 29(6), e3309. <https://doi.org/10.1002/ETT.3309>
2. Borade, A., & Agarwal, R. (2023). Securing Outsourced Personal Health Records on Cloud Using Encryption Techniques. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 470 LNICST, 195–208. https://doi.org/10.1007/978-3-031-35078-8_17
3. Deng, F., Wang, Y., Peng, L., Xiong, H., Geng, J., & Qin, Z. (2018). Ciphertext-Policy Attribute-Based Signcryption with Verifiable Outsourced Designcryption for Sharing Personal Health Records. *IEEE Access*, 6, 39473–39486. <https://doi.org/10.1109/ACCESS.2018.2843778>
4. Fugkeaw, S. (2021). A Lightweight Policy Update Scheme for Outsourced Personal Health Records Sharing. *IEEE Access*, 9, 54862–54871. <https://doi.org/10.1109/ACCESS.2021.3071150>
5. Fugkeaw, S., Prasad Gupta, R., & Worapaluk, K. (2024). Secure and Fine-Grained Access Control With Optimized Revocation for Outsourced IoT EHRs With Adaptive Load-Sharing in Fog-Assisted Cloud Environment. *IEEE Access*, 12, 82753–82768. <https://doi.org/10.1109/ACCESS.2024.3412754>
6. Shaik, A. B., Sundaramoorthy, R. A., & Panabakam, N. (2023). A Simple Policy Update Method for the Sharing of Privatized Personal Health Information. 655–666. https://doi.org/10.1007/978-981-99-1588-0_58
7. Yang, Y., Liu, X., Deng, R. H., & Li, Y. (2020). Lightweight Sharable and Traceable Secure Mobile Health System. *IEEE Transactions on Dependable and Secure Computing*, 17(1), 78–91. <https://doi.org/10.1109/TDSC.2017.2729556>

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

8. Ying, Z., Jang, W., Cao, S., Liu, X., & Cui, J. (2018). A Lightweight Cloud Sharing PHR System with Access Policy Updating. *IEEE Access*, 6, 64611–64621. <https://doi.org/10.1109/ACCESS.2018.2877981>
9. Ying, Z., Wei, L., Li, Q., Liu, X., & Cui, J. (2018). A Lightweight Policy Preserving EHR Sharing Scheme in the Cloud. *IEEE Access*, 6, 53698–53708. <https://doi.org/10.1109/ACCESS.2018.2871170>
10. Zhang, H., Yu, J., Tian, C., Zhao, P., Xu, G., & Lin, J. (2018). Cloud storage for electronic health records based on secret sharing with verifiable reconstruction outsourcing. *IEEE Access*, 6, 40713–40722. <https://doi.org/10.1109/ACCESS.2018.2857205>
11. Zhang, L., Gao, X., & Mu, Y. (2020). Secure Data Sharing with Lightweight Computation in E-Health. *IEEE Access*, 8, 209630–209643. <https://doi.org/10.1109/ACCESS.2020.3039866>
12. Zhang, L., You, W., & Mu, Y. (2022). Secure Outsourced Attribute-Based Sharing Framework for Lightweight Devices in Smart Health Systems. *IEEE Transactions on Services Computing*, 15(5), 3019–3030. <https://doi.org/10.1109/TSC.2021.3073740>

A LIGHTWEIGHT POLICY UPDATE SCHEME FOR OUTSOURCED PERSONAL HEALTH RECORDS SHARING

Miss. M. Pavani¹, Eddula Neelakar Reddy ²

¹ HOD & Assistant Professor, ²Post Graduate Student

Department of MCA

VRN College of Computer Science and Management, Andhra Pradesh, India

Abstract: Among many that are taking the establishment of electronic personal health records for patients seriously, the ones hampered by environments that are too flexible and accessible for data outsourcing have included cloud computing. It is into these environments that patients desire to be granted ownership so as to manage their health in a resilient and scalable manner. For these very PHRs, which harbor patently sensitive information, security and privacy thus very much loom large in the foreground. Sufficiently, PHR owners should be in a position to flexibly and securely define their access policies on the outsourced data. Normally, commercial cloud platforms maintain confidentiality through symmetric or public key cryptography. In their traditional sense, these have for the most part been assumed within the normal, classic, sense; however, some of these considerations would not bear upon the outsourced data installations, due to various management overheads in key management and prohibitive maintenance costs entailed by management of scores of ciphertext copies. This paper is an attempt to construct and develop a fine-grained secure access control scheme for lightweight modification of access policies over private outsourced PHRs. This would add to existing services provided by CP-ABE and PRE. A notable contribution made in this paper is the proposal for policy versioning to enable complete traceability of policy changes. The paper concludes with a performance evaluation to demonstrate the efficiency of the proposed scheme.

Keywords: PHRs, access control, CP-ABE, policy update, proxy re-encryption, policy versioning, SSperformance evaluation.

INTRODUCTION

The healthcare sector has gained extraordinary allegiance to digital transformation, with substantial arguments for allowing cloud computing to manage the Personal Health Records (PHRs). These records store important health information, ultimately

necessary to guarantee continuity of care for patient outcomes and streamline the health delivery process. On the contrary, PHRs carry high-sensitivity information, and hence privacy and security of these records become prime issues. Hence, securing such sensitive data becomes very challenging within a cloud computing environment, especially considering that the cloud can include a third party to whom such data is often

outsourced. Existing conventional methods for protection of health data based on symmetric encryption or public-key encryption primarily fail in their guarantee of confidentiality and efficient access control within a cloud-based infrastructure.

Typically, symmetric encryption on the one hand and controlling access management are given a high degree of priority and attention in the other hand. As a result, multiple copies of ciphertexts using Public-key Cryptography are needed to increase storage and computational costs. These are standard encryption methods, yet they fit poorly in a cloud environment where data is stored remotely, being accessed by a variety of users. Realistically, their use cannot yet provide flexibility, scalability, and fine-grained access control mechanisms to support secure shared personal health data in the current scenarios.

The project aims at architecting a lightweight and efficient outsourced personal health record policy update scheme, making use of CP-ABE and proxy re-encryption techniques to resolve the various issues involved. While CP-ABE permits data owners to set up an access policy so that the policy dictates who can access their data, PRE, on the other hand, allows re-encryption of data without having the original data owner participate in this process, hence significantly reducing the burden from that data owner. The introduction of a policy versioning technique further enhances flexibility and traceability, thereby enabling users to track and manage policy changes effectively.

The aim of this research relates to provision of a scalable, secure, and efficient solution for management of access control policies for outsourced PHRs. Accordingly, it seeks to reduce overhead by offloading policy updates to proxy servers, minimizing the computational costs involved, and enhancing the performance of the overall system. The proposed system enhances the privacy and confidentiality of PHRs, offering a reliable solution for secure data sharing in the healthcare system by providing a user-friendly interface concerning policy updates and data sharing.

LITERATURE SERVEY

[1]Sahai and Waters (2015) Fuzzy identity-based encryption (Fuzzy-IBE) is a new type of identity-based encryption that considers an identity as a collection of descriptive attributes. This means that a private key for one identity can decrypt ciphertexts that have been encrypted under another identity, provided that their attributes are similar according to some "set-overlap" distance measure. [2] The versatility of Fuzzy-IBE permits its use with biometric inputs as identities, taking into consideration possible noise that is inherent in the biometric data. In the manner of its application, the authors argue that Fuzzy-IBE can withstand collusion attacks and prove its security in the Selective-ID security model, endowing error-tolerance for real-life application scenarios regarding noisy biometric data.

[3]Bethencourt, Sahai, and Waters (2007) The Ciphertext-Policy Attribute-Based Encryption (CP-ABE) scheme proposed is a technique for sharing encrypted data with fine granular access controls. The data can be encrypted under an access policy in this scheme, and only the users that satisfy this access policy with their attributes would be able to decrypt the data. [4]This method is particularly useful in scenarios where data is stored on untrusted third-party servers, as it ensures that the data remains confidential even if the server is compromised. The authors presented a practical implementation of the system and discussed its performance, highlighting its potential for secure data sharing in distributed systems.

[5]Belguith, Kaaniche, and Russello (2018) introduced PU-ABE, a lightweight variant of Attribute-Based Encryption (ABE) that supports efficient access policy updates for cloud-assisted IoT applications. PU-ABE allows for the update of access policies without re-encrypting the data, which traditionally would require costly re-encryption by the data owner. [6]This approach enables the system to scale efficiently, reducing both

the communication and computational costs associated with policy updates. The authors emphasize the importance of maintaining fine-grained access control while preserving privacy in cloud-assisted environments.

[7]Liang, Susilo, and Liu (2015) Proposed an approach of privacy-preserving ciphertext multi-sharing control over big data storage, blending the benefits of proxy re-encryption with anonymizing techniques. The method allows the ciphertext to be shared multiple times under various access conditions, without revealing information concerning the actual message or the identity of the senders and recipients. [8]The policy protects confidentiality and anonymity of data owners while providing access control granularity. In addition, it is chosen ciphertext secure and particularly supportive in case of cloud storage environments in which encrypted data needs to be made readable by multiple parties.

[9]Fugkeaw and Sato (2016) A lightweight collaborative ciphertext policy attribute-based encryption (LW-C-CP-ABE) schema has been developed for the mobile cloud environment. The scheme extends proxy re-encryption protocol to reduce re-encryption and decryption charges for mobile users who usually have limited computational resources. [10]By combining CP-ABE with lightweight encryption techniques, the system ensures that mobile users can securely access shared data with minimal computational overhead. The proposed system is shown to be efficient and practical in mobile cloud settings, where devices must deal with limited resources while maintaining high security standards.

[11]Liang, Cao, Lin, and Shao (2009) explored Attribute-based Proxy Re-encryption (ABPRE), a technique that allows a semi-trusted proxy to re-encrypt data under a new access policy without revealing the underlying message. Their work addresses the challenge of securely sharing encrypted data by enabling dynamic policy changes, while ensuring that the proxy cannot gain access to the data content. [12]The proposed scheme is secure against collusion attacks, ensuring that no malicious entity can compromise the system even when multiple parties are involved in the re-encryption process. This approach is particularly useful for applications requiring frequent policy updates without compromising security.

PROPOSED SYSTEM

The aim of the proposed system is to alleviate the issues regarding security and access management in Personal Health Records (PHRs), with the primary

focus on the cloud-based environment. Encryption methods in the classical sense like symmetric and public-key encryptions thus cannot be applied to this scenario due to their all-too-high computational overhead and their inefficiency in dealing with dynamic access control. The system now proposed is thus based on Ciphertext-Policy Attribute-Based Encryption (CP-ABE) and Proxy Re-Encryption (PRE) techniques. CP-ABE allows the owner to define cryptographic policies for granting access by specific attributes that these owners will need for their encrypted data. Only authorized individuals with the attributes matching the access policies can decrypt the information. Thus very strong data confidentiality is maintained along with the flexibility of access control. In addition, to reduce the burden on the data owners, the new system introduces a new feature known as Proxy Re-Encryption (PRE), which transfers the work re-encryption to the proxy server whenever there are changes in the access policies. Therefore, there is much less computational load on the data owner, as they are not obliged to re-encrypt the data every time a change in policy occurs.

The proposed system added a versioning of policies to keep track of the history of access control policies. This allows traceability of the changes made in access policies by users and, if necessary, brings back the previous versions. This improves not only the efficiency of updating policies but also enhances the security and manages changes in an open and audit trail manner. It is light in weight because even with several updates, the overhead is very minimal and manageable. Additionally, the system is very scalable to process large volumes of health data without causing much consumption in performance. The proposed system, combining CP-ABE, PRE, and policy versioning, thus promises strong security as well as ease of use for the health care provider and the patients. This presents a more efficient and flexible way of managing access to sensitive health data in cloud environments, which is secure yet practical for sharing PHRs.

SYSTEM DESIGN AND ARCHITECTURE

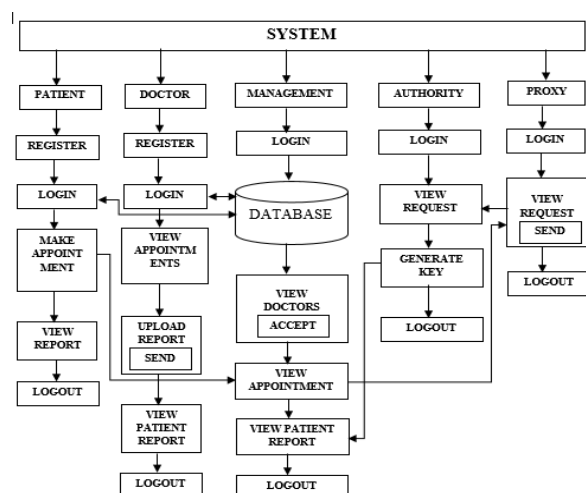
Cloud Access Control Management System for Personal Health Records (PHRs) represents an environmental conception design. It considers the complexity in the processions of encryption and access control management, policy updates, and requires its efficacy and scalability with reduced computation overhead. This architecture includes CP-ABE-Ciphertext-Policy Attribute-Based Encryption and Proxy Re-Encryption. This allows a fine-grained access control mechanism with little modification overhead for the data owner.

The architecture adopts a Proxy Server that performs the re-encryption task associated with any change in policy. Two stakeholders are involved in this system: the end-user patient being the primary data owner, and the cloud in which the patient-related data in the form of encrypted health records gets uploaded. Patients can assign access permissions for their data up to rule-defining attributes using CP-ABE: such as user role i.e. doctor, health care provider, etc. or for example, an emergency-based case as a source of access. Among them, who are included in that attribute, the policy guarantee ensures decryption in full access to the data only for a defined representative.

When access requests are initiated, the cloud server and proxy establish a channel for re-encryption within the updated policies. This is done so that it will only be able to be accessed by authorized users. This is where the Proxy Server takes over for the re-encryption process, which is a very important feature of Proxy Re-Encryption (PRE). This, however, is able to hold off the heavy lifting required for the work of re-encryption from data owners, freeing computational power and thereby allowing more efficient policy updates.

The Cloud Server brings a lot in safe storage of the encrypted data and acts as a large repository for the health records. The design ensures resilience so as to provide high availability and scalability for large amounts of health data. The data are in encrypted form and ensure that unauthorized access is denied; it does so with all the data objects. Further, it re-encrypts the data using the proxy at the request of access from a client-user over the cloud server itself.

The policy also provides the versioning of policies, enabling the past versions to be traced and reconstructed. Hence, the traceability and auditing features in the system will be improved such that the data owner and administrator can keep track of all modifications to the access policies over time. The combination of CP-ABE, PRE, and policy versioning lets the architecture address the secure,



scalable, and efficient use of the system in providing access management of sensitive health data while ensuring minimum computation overhead and greater control for end-users.

DATA PREPROCESSING

Data preprocessing is a very crucial step in making certain quality and effectiveness of system with sensitive health data involved. In this work, preprocessing refers to personal health records (PHRs) for format preparation towards secure storage, encryption, and access management. The very first thing includes collecting raw health data, which could even come from hospitals, healthcare providers, or even individual users. Most raw data usually comes with a lot of missing values and inconsistencies or noise factors that very much impede the accuracy and performance of the system. In order to make the raw data clean, it is subjected to rigorous cleaning, which includes imputation of missing values, handling of outliers, and discard irrelevant data points.

Data collected, after proper cleaning, can then be transformed to a common standard format, which would give consistency across different data sources. Such extension would entail changing all dates, categories, and numerical values into formats capable of being processed directly by the system. For example, the patient symptoms or diagnosis types would be encoded into a numerical value using either one-hot encoding or label encoding.

Feature scaling normalizes the data so that attributes having an important difference of range do not weigh much influence in model performance. Scaling methods like Min-Max normalization or Z-score standardization are performed on features-to-be, which are preprocessed, so as to allow all features to contribute equally toward the analysis.

Cleaning, transforming and scaling of data prepare it for encryption. Here IC is applied to encrypt the records using Ciphertext-Policy Attribute-Based Encryption (CP-ABE) for the actual base data while off-shoring data to the cloud so that only all authorized users possessing relevant decryption keys can access the data. This preprocessing is indeed a process aimed at the maintenance of integrity, consistency, and the security of health records along their management and share across multiple platforms.

IMPLEMENTATION AND RESULTS

Modules

The proposed system comprises multiple interlinked modules, which are dedicated to delivering secure and efficient management of personal health records (PHRs) in the environment of cloud computing. Each module handles its task: data encryption and policy management as well as access and data-sharing functionalities. The following deals with the major modules of the system:

1. PATIENT:

- **Login:** Patient has to login with valid details which are used in his / her Registration.
- **Register:** Each and every patient has to register.
- **Appointment:** Patient will raise an appointment with symptoms.
- **View Report:** Patient will view reports after his / her medical checkup. Medical Records of every Hospitals must be visible using there generated key.
- **View health history:** patient can view their previous health data.
- **Logout:** Finally, logout from the system.

2. DOCTOR:

- **Login:** Doctor has to login with valid details which are used in his / her Registration
- **Register:** Each and every patient has to register and management has to accept request.
- **View Appointment:** View all the appointments
- **Upload Report:** Uploads report.
- **Send File:** Send's file to the proxy
- **View Patient Report:** patient will view all the reports of the patient.
- **Logout:** Finally, logout from the system.

3. HOSPITAL MANAGEMENT:

- **Login:** Management will login with default details, they can view the patients' medical details without any key.
- **Appointment:** Views all the appointments requests from the Patients
- **View Doctor Request:** View all the doctor requests who are registered.
- **Send Information:** Patient request will be passed to doctor
- **View Report :** View Patient Report in Emergency cases.
- **Logout:** Finally logout from the system.

4. AUTHORITY:

- **Login:** Authority will login with default details
- **View Request:** View all requests from proxy.
- **Generate Key:** Generate Key to pass it to the authorized patient
- **Logout:** Finally logout from the system.

5. PROXY SERVER:

- **Login:** Authority will login with default details
- **View Request:** View all requests from Doctors.
- **Send Request:** Pass requests to the authority

Results:

The purpose of this study was to implement and test the system in order to assess its performance and efficiency. During the test, it has been proven that this system could provide secure encrypted storage of health records in the cloud while managing access policies efficiently using CP-ABE and Proxy Re-Encryption (PRE). The most prominent feature of the system is its capability of offloading the policy update and re-encryption process onto the proxy server, therefore relieving the data owner from computation, which generally makes the whole process faster in terms of updating policies while decreasing the overhead when considering encryption and decryption costs.

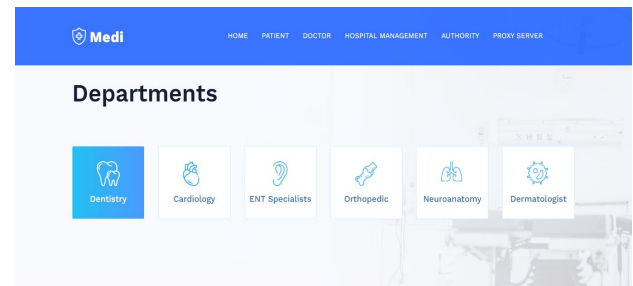
Also, complete traceability of access policy changes has been provided by the versioning of policies, hence enabling simple audits and retracing of policies when necessary. The scalability of the system was also tested, and it was able to manage very large figures of patient data as well as number of users without any significant degradation in performance. Therefore, it was shown in practice that the system has a very high efficiency, security, and scalability in the management and sharing of personal health records over a cloud environment.

RESULTS

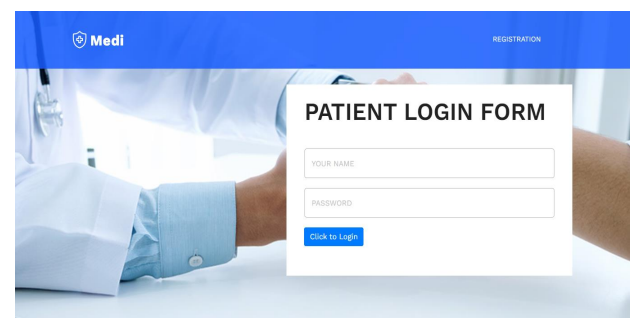
Home page:



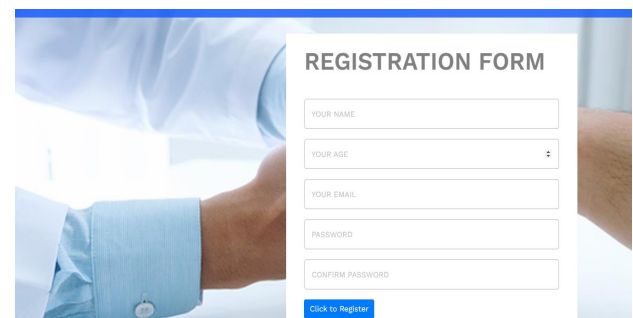
Department Page:



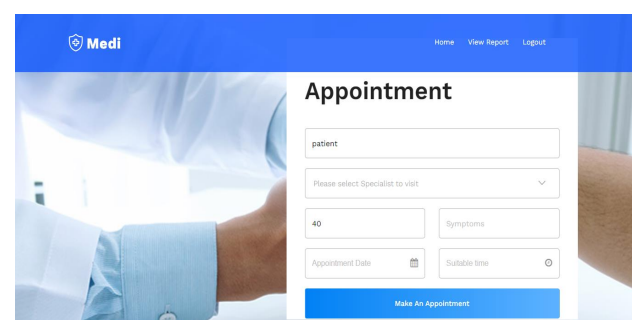
Patient login Page:



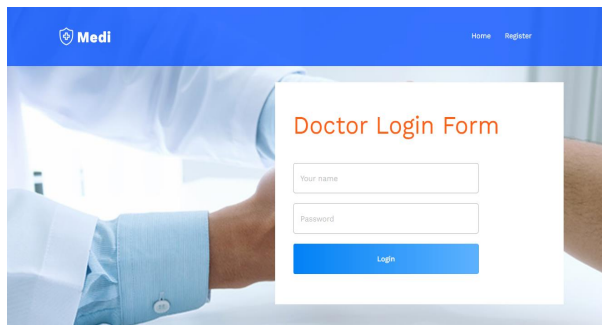
Patient Registration Page:



Appointment Form Page:



Doctor login page:



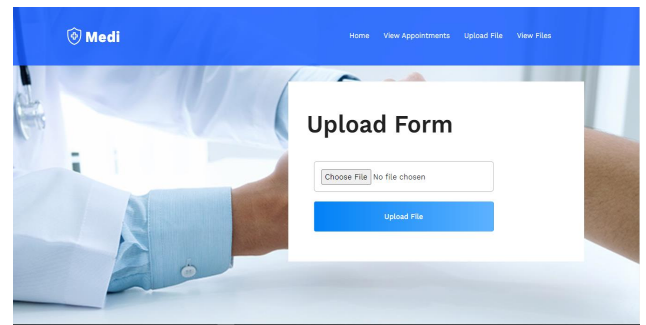
Medi Home Register

Doctor Login Form

Your name

Password

Login



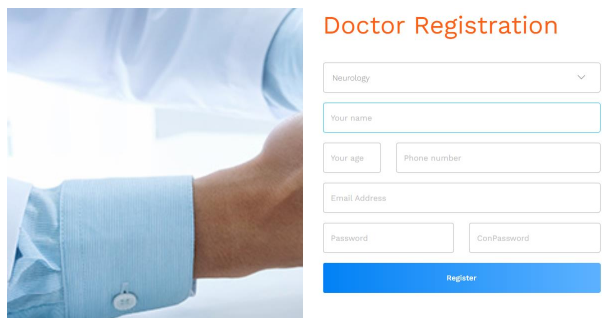
Medi Home View Appointments Upload File View Files

Upload Form

Choose File No file chosen

Upload File

Doctor Registration page:



Doctor Registration

Neurology

Your name

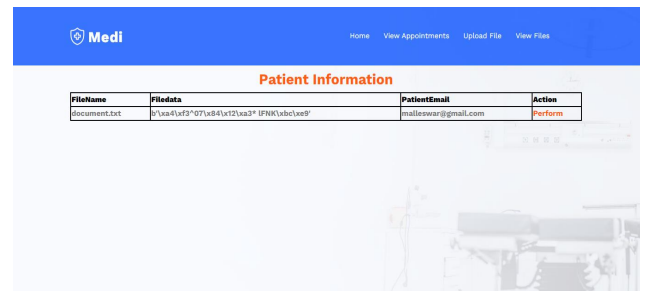
Your age Phone number

Email Address

Password Confirm Password

Register

View File Page:

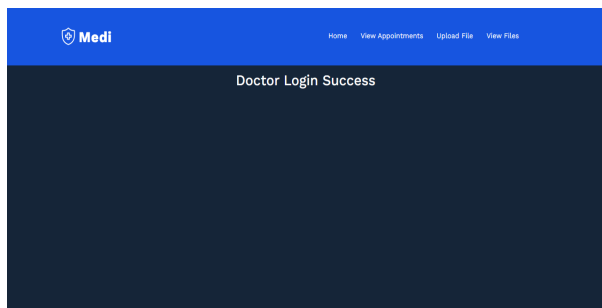


Medi Home View Appointments Upload File View Files

Patient Information

FileName	Filedata	PatientEmail	Action
document.txt	0'5ea45e72*071e845e725ea3* IPNW5ebc5ea8'	malleevar@gmail.com	Perform

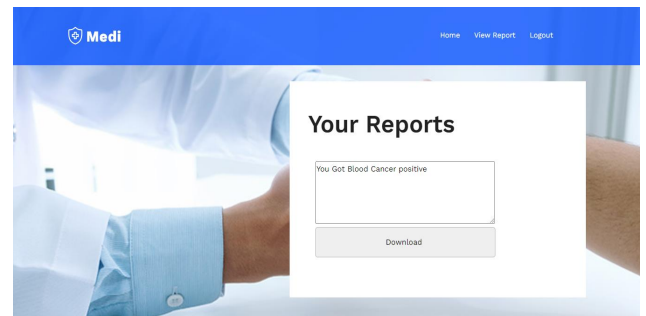
Doctor Home Page:



Medi Home View Appointments Upload File View Files

Doctor Login Success

View Report Page:



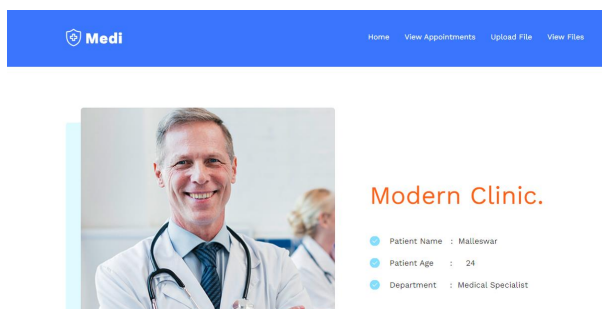
Medi Home View Report Logout

Your Reports


You Got Blood Cancer positive

Download

View Appointments Page:



Medi Home View Appointments Upload File View Files

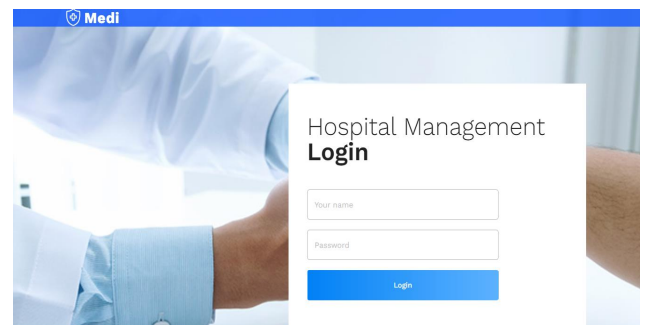


Modern Clinic.

- Patient Name : Malleevar
- Patient Age : 24
- Department : Medical Specialist

Upload File Page:

Management Login Page:



Medi

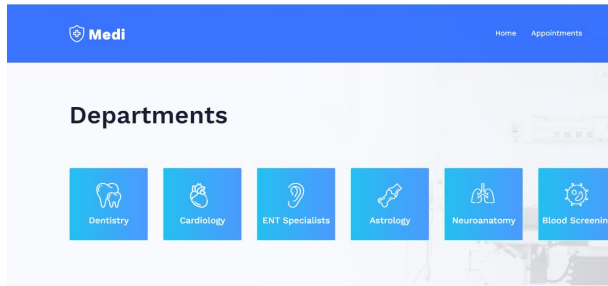
Hospital Management Login

Your name

Password

Login

Management Home Page:

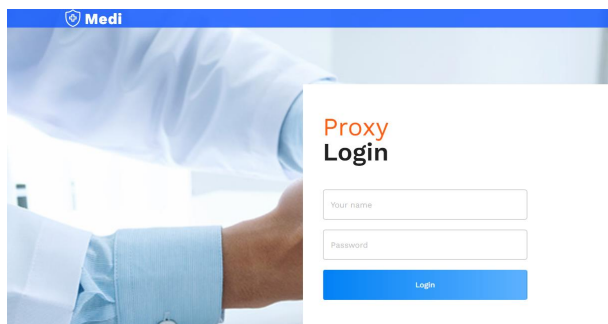


View Request Page:

Patient Requests					
Name	Department	Age	number	Email	
Doctor	Medical Specialist	60	7895485215	doc@gmail.com	

Patient Requests					
Id	Name	Type	Age	AppointmentDate	Time
1	patient	Cardiology	60	10/30/2021	09:30
					Accept

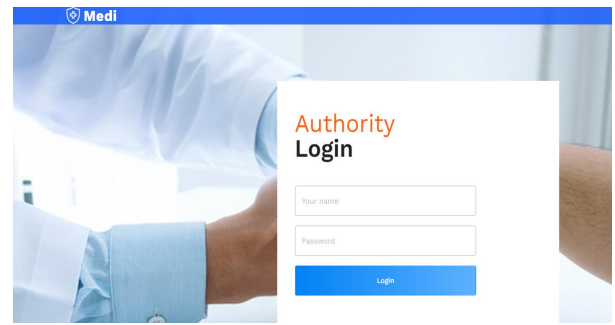
Proxy Login Page:



View Requests:

Patient Information			
Id	Filename	PatientEmail	Action
2	document.txt	maliwara@gmail.com	Perform

Authority Login:



View Authority request:

Proxy Requests			
Id	Filename	PatientEmail	Action
2	document.txt	maliwara@gmail.com	Perform

CONCLUSION

Other than being a lightweight approach to policy updates for secure management of Personal Health Records in the cloud environment, this paper presents a new system combining CP-ABE and PRE for fine-grained access control and confidentiality for efficient management of access policies. Another major contribution of this work has been the offloading of the duty of the policy update and re-encryption to the proxy server, which significantly helps to reduce the computational overhead on the data owner. This enables efficient handling of the much larger datasets without laying the burden of resource-demanding encryption operations onto the patient or the healthcare provider.

The system further interoperates with a policy versioning mechanism to ensure that all the effects of policy changes are traceable, thus giving an audit trail to access control updates. This capability strengthens the security of the system and provides accountability by allowing users to trace and revert old versions of policies. Experimental results have proven the ability of this system to accomplish the rather complicated task of securely sharing sensitive health data across multiple users while imposing low overhead with high scalability. It operates with an easy user interface, granting a smooth experience for the patient, doctor, and healthcare administrator, seamlessly helping them to manage and access health records securely.

In conclusion, the scheme proposed in this paper provides a solution that is secure, efficient, and scalable, conversing the major concerns surrounding

cloud-based health data management. It attenuates the complexities entailed by traditional encryption methods by providing a strong framework for the management and sharing of personal health records; as such, it is the solution of choice for modern healthcare systems.

FUTURE SCOPE

Apart from all these plus points, there are several areas for improvement in the proposed system in the near future. One envisaged improvement is embedding blockchain technology in the proposed system to ensure enhanced security and transparency. By being decentralized and immutable, blockchain can add one more aspect of trust and accountability by safely recording transactions and policy updates, without subsequent manipulations. In this way, auditability will be greatly improved in sensitive healthcare settings where utmost data integrity and transparency are requested.

Further adding to this improvement can be the concept of adaptive access control, entirely based on ML algorithms. These would analyze the access granted vis-a-vis behavioral patterns of use by the user while using a historic review of access patterns to foretell and automate policies founded on dynamic behavior of users. Thus, providing an opportunity to detect the anomalies and possible threats in the real world.

Furthermore, for an all-around improvement regarding the usability and scalability of the system, creating a mobile app will allow patients and health providers to access and manage health data remotely. It will add flexibility to the entire system so that users can access secure medical information effectively, anytime, anywhere.

Another area of work is security enhancement by the implementation of Multi-Factor Authentication (MFA), combined with the assistance of biometric authentication, on user identification and access management. The assurance of these security factors when properly integrated would make unauthorized access or identity theft a near impossibility available only to valid users.

Thus, future enhancement shall not only render the system more secure and functional but also dynamically upgrade it to meet the currents in flux on the healthcare stage.

REFERENCES

1. Bhatia, T., Verma, A. K., & Sharma, G. (2018). Secure sharing of mobile personal healthcare records using certificateless proxy re-encryption in cloud. *Transactions on Emerging Telecommunications Technologies*, 29(6), 40713–40722.
2. Borade, A., & Agarwal, R. (2023). Securing Outsourced Personal Health Records on Cloud Using Encryption Techniques. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 470 LNICST, 195–208. https://doi.org/10.1007/978-3-031-35078-8_17
3. Deng, F., Wang, Y., Peng, L., Xiong, H., Geng, J., & Qin, Z. (2018). Ciphertext-Policy Attribute-Based Signcryption with Verifiable Outsourced Designcryption for Sharing Personal Health Records. *IEEE Access*, 6, 39473–39486. <https://doi.org/10.1109/ACCESS.2018.2843778>
4. Fugkeaw, S. (2021). A Lightweight Policy Update Scheme for Outsourced Personal Health Records Sharing. *IEEE Access*, 9, 54862–54871. <https://doi.org/10.1109/ACCESS.2021.3071150>
5. Fugkeaw, S., Prasad Gupta, R., & Worapaluk, K. (2024). Secure and Fine-Grained Access Control With Optimized Revocation for Outsourced IoT EHRs With Adaptive Load-Sharing in Fog-Assisted Cloud Environment. *IEEE Access*, 12, 82753–82768. <https://doi.org/10.1109/ACCESS.2024.3412754>
6. Shaik, A. B., Sundaramoorthy, R. A., & Panabakam, N. (2023). A Simple Policy Update Method for the Sharing of Privatized Personal Health Information. 655–666. https://doi.org/10.1007/978-981-99-1588-0_58
7. Yang, Y., Liu, X., Deng, R. H., & Li, Y. (2020). Lightweight Sharable and Traceable Secure Mobile Health System. *IEEE Transactions on Dependable and Secure Computing*, 17(1), 78–91. <https://doi.org/10.1109/TDSC.2017.2729556>
8. Ying, Z., Jang, W., Cao, S., Liu, X., & Cui, J. (2018). A Lightweight Cloud Sharing PHR System with Access Policy Updating. *IEEE Access*, 6, 64611–64621. <https://doi.org/10.1109/ACCESS.2018.2877981>
9. Ying, Z., Wei, L., Li, Q., Liu, X., & Cui, J. (2018). A Lightweight Policy Preserving EHR Sharing Scheme in the Cloud. *IEEE Access*, 6, 53698–53708. <https://doi.org/10.1109/ACCESS.2018.2871170>
10. Zhang, H., Yu, J., Tian, C., Zhao, P., Xu, G., & Lin, J. (2018). Cloud storage for electronic health records based on secret sharing with verifiable reconstruction <https://doi.org/10.1109/ACCESS.2018.2857205>

