CSCI 531 PROJECT REPORT

Name: Pradyot Omprakash Chhatwani USC ID: 2850400993

I. SYSTEM WORKFLOW

The workflow for a secure server-client communication system for medical audit data is as follows:

1. **System Initialization:**

   - The server initializes user data management by loading a JSON file containing hashed user credentials or creating an empty dictionary if the file doesn't exist.
   - The server sets up encryption and hashing functions for secure communication.
   - An AuditServer instance is created for handling auditing and integrity checks of medical data.
   - The server sets up a socket to listen for incoming client connections.
   - A separate thread is initiated to check the server's database's integrity continuously.

2. **Client Connection and Authentication:**

   - The client connects to the server and receives the server's RSA public key.
   - The client creates pre-master secret, encrypts it using server's public key, and sends it to server.
   - The server generates an RSA key pair, creates an AES-GCM cipher using the pre-master secret, and authenticates the client's username and password.

3. **User Registration (if necessary):**

   - If the client is not registered, the server prompts the client to enter a password and user type (patient or auditor) for registration.

4. **Data Integrity Check:**

   - A separate thread is started on the client side for continuously receiving data integrity messages from the server.

5. **Client-Server Interaction:**

   - After successful authentication and registration, the client and server enter loops where they continuously send requests and process responses.
   - The client interacts with the server using an AuditClient instance, which handles user interaction based on the user type.
   - The server processes client requests using the AuditServer instance, which interacts with the MedicalDataSystem to perform operations on medical records.

6. **Medical Data and Blockchain Interaction:**

   - The MedicalDataSystem class handles medical records and uses blockchain for data integrity.
   - CRUD operations and other operations on medical records are performed using methods like create_record, delete_record, update_record, get_records, print_record, and copy_record.
   - The query_user_logs method allows querying logs based on user type and patient ID.
   - The check_integrity method checks the integrity of the blockchain.

7. **Blockchain Management:**

- The BlockChain class represents entire blockchain, maintaining integrity of medical records.
- The Block class represents a single block in the blockchain, with relevant attributes and a method for calculating its hash.
- The BlockchainFile class handles storage and retrieval of the blockchain to and from a file.
- This workflow ensures secure communication, user authentication, and data integrity in a medical audit system. The client and server interact to perform various operations on medical records, and the blockchain is used to maintain the integrity of the data.

II. SYSTEM ARCHITECTURE

A. SYSTEM COMPONENTS:

1. Authentication Server:

This component, included in Server.py, handles user authentication by verifying credentials and managing user registration. It stores credentials in a JSON file using SHA-256 hashing.

2. Audit Server:

The AuditServer class in AuditServer.py processes client requests on the server side. It interacts with the MedicalDataSystem to perform operations on the medical records, ensuring proper auditing and integrity checks are in place.

3. Medical Data System:

The MedicalData class in MedicalDataSystem.py manages the medical records and interacts with the blockchain to maintain the integrity of the medical data. It provides methods for creating, deleting, updating, retrieving, and querying records and user logs.

4. Blockchain System:

The BlockChain.py file implements a blockchain system, which is used to maintain the integrity of the medical records within the medical audit system. It consists of three classes: Block, Blockchain, and BlockChainFile.

5. Client:

The Client.py code manages communication with the server from the client side. It establishes a secure connection and sends user credentials for authentication. Once authenticated, it can access or modify data as per user requests.

6. Audit Client:

The AuditClient class in AuditClient.py handles user interaction on the client side, providing a menu of available options based on the user type (patient or auditor). It constructs JSON objects representing user requests and processes server responses accordingly.

B. COMMUNICATION PATTERNS:

1. Client-Authentication Server Communication:

The client establishes a connection with the authentication server and sends its credentials (username and password) for authentication. The server verifies the credentials and responds with a successful login or an error message.

2. Client-Audit Server Communication:

After successful authentication, the client interacts with the Audit Server by sending requests and receiving responses. The communication between the client and server is encrypted using RSA and AES-GCM. The Audit Server processes the client's requests and returns the results or error messages.

3. Audit Server-Medical Data System Communication:

The Audit Server communicates with the Medical Data System by calling methods on the MedicalData class. These methods perform CRUD operations and other actions on the medical records.

4. Medical Data System-Blockchain System Communication:

The Medical Data System interacts with the blockchain system by calling methods on the Blockchain class. These methods include adding new blocks, querying user logs, and checking the integrity of the blockchain.

The system's components work together to enable secure server-client communication for a medical audit use case. The communication patterns involve various components interacting with one another to authenticate users, process client requests, manage medical records, and maintain the integrity of the medical data through a blockchain system. The system employs cryptographic techniques, such as RSA and AES-GCM, to ensure secure data transfer between the client and server.

III. CRYPTOGRAPHIC COMPONENTS

A. CRYPTOGRAPHIC PRIMITIVES

The system uses several cryptographic primitives to ensure secure communication and data handling. These cryptographic components play a crucial role in achieving the goals of privacy, identification and authorization, queries, immutability, and decentralization.

1. AES-GCM:

AES-GCM (Advanced Encryption Standard-Galois/Counter Mode) is a widely used block cipher operation that provides confidentiality and data integrity. It is commonly used in network protocols, such as TLS, IPsec, and SSH.

AES-GCM combines the AES block cipher with the Galois/Counter Mode (GCM) of operation. AES is a symmetric block cipher that encrypts and decrypts data in fixed-size blocks of 128 bits.

GCM is a mode of operation that provides confidentiality and integrity using a counter-based approach.

AES-GCM encrypts the data in blocks of 128 bits using AES in counter mode (CTR). The resulting ciphertext is then authenticated using a message authentication code (MAC) based on Galois field multiplication. The authentication tag is appended to the ciphertext to provide integrity and authentication of the encrypted data.

The security of AES-GCM depends on the key size and the non-repeating nature of the counter used in the CTR mode. AES-GCM provides strong security and is widely used in many security protocols due to its high performance and security features.

In this system, AES-GCM is used to encrypt and decrypt data transferred between the server and clients, ensuring privacy and data integrity during communication.

2. **RSA:**

RSA (Rivest–Shamir–Adleman) is a public-key cryptography algorithm widely used for secure communication over the Internet. It was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and is named after their last names.

RSA is based on the idea that it is hard to factorize the product of two large prime numbers. The algorithm generates a public key and a private key, where the public key can be shared with anyone, while the owner keeps the private key secret.

To encrypt a message using RSA, the sender uses the recipient's public key to transform the message into an unintelligible form that can only be decrypted using the recipient's private key. To decrypt the message, the recipient uses their private key to transform the encrypted message back into its original form.

RSA is widely used for secure communication, including online transactions, email encryption, and secure remote access. It is considered a very secure algorithm, although it can be vulnerable to attacks if the key length is too short or if the implementation is flawed.

This system uses RSA for key exchange between the server and clients. The server generates an RSA key pair and sends the public key to the client. The client uses the server's public key to encrypt a pre-master secret, which is then sent to the server. The server decrypts the pre-master secret using its private key and creates an AES-GCM cipher for further communication. This approach ensures secure key exchange while preventing man-in-the-middle attacks.

3. **SHA-256:**

SHA-256 is a cryptographic hash function widely used in various security applications, such as digital signatures, password storage, and blockchain technology.

A hash function is a mathematical algorithm that takes input data of arbitrary size and generates a fixed-size output called a hash or a digest. In the case of SHA-256, the output size is 256 bits.

To produce the hash value, the SHA-256 algorithm takes the input data and performs a series of operations, including bitwise logical operations, modular arithmetic, and bitwise rotations. The

resulting hash value is unique to the input data and is a digital fingerprint of that data. Even a small change in the input data will result in a completely different hash value.

SHA-256 is considered a secure hash function because it is computationally infeasible to find two different inputs that produce the same hash output. This property is known as collision resistance. It is also computationally infeasible to reverse engineer the original input data from the hash value, making it a one-way function.

In this system, SHA-256 is used for hashing user credentials and blocking contents in the blockchain. The hashing of user credentials ensures secure storage and prevents unauthorized access to user data. The hashing of block contents in the blockchain guarantees data integrity and immutability.

The cryptographic primitives mentioned above support the following goals:

1. **Privacy:**

   AES-GCM and RSA provide secure communication channels between the server and clients, ensuring data privacy during data transfer.

2. **Identification and Authorization:**

   Hashed user credentials and secure key exchange via RSA enable the system to authenticate users and authorize access to medical data.

3. **Queries:**

   The system allows users to query medical records and logs securely, as data transmitted between the server and clients is encrypted using AES-GCM, ensuring query privacy.

4. **Immutability:**

   SHA-256 hashing of block contents in the blockchain guarantees that any changes to the data will result in different hashes, ensuring data immutability. The integrity check thread continually monitors the integrity of the server's database and informs the client in case of any discrepancies.

5. **Decentralization:**

   While the system does not implement a fully decentralized architecture, the blockchain's use provides a decentralized data integrity mechanism, ensuring that no single point of failure can compromise the medical records' security.

B. CONCRETE ENCRYPTION SCHEMES AND KEY MANAGEMENT APPROACHES

The concrete encryption schemes and key management approaches used in the system include:

1. **RSA Key Generation and Exchange:**

   The server generates an RSA key pair and shares the public key with the client. The client then generates a pre-master secret, encrypts it using the server's public key, and sends it back to the

server. The server uses its private key to decrypt the pre-master secret and establishes an AES-GCM cipher for further communication.

2. **AES-GCM Encryption and Decryption:**

   After the key exchange, the server and clients use the AES-GCM cipher for encryption and decryption. AES-GCM ensures confidentiality, integrity, and authenticity during data transfer.

3. **SHA-256 Hashing:**

   User credentials are hashed using SHA-256 before being stored in the server's database. Additionally, SHA-256 is used to hash the contents of the blocks in the blockchain, ensuring data integrity and immutability.

In conclusion, the cryptographic components used in this secure server-client communication system support the goals of privacy, identification and authorization, queries, immutability, and decentralization. The use of AES-GCM, RSA, and SHA-256 provides a robust combination of cryptographic primitives that work together to ensure secure data transfer, storage, and integrity.

1. **Secure Data Transfer:**

   The use of RSA for key exchange and AES-GCM for data encryption and decryption ensures that data transferred between the server and clients are secure and protected against eavesdropping and man-in-the-middle attacks.

2. **Secure Data Storage:**

   The system hashes user credentials using SHA-256 before storing them, protecting sensitive information against unauthorized access and data breaches. This approach adds an additional layer of security to user authentication and authorization processes.

3. **Data Integrity and Immutability:**

   The implementation of a blockchain with SHA-256 hashing guarantees that the medical records and user logs are tamper-proof, as any alteration to the data will result in a different hash, making it easy to detect unauthorized changes. The integrity check thread monitors the server's database continuously, alerting the client if any integrity issues are detected.

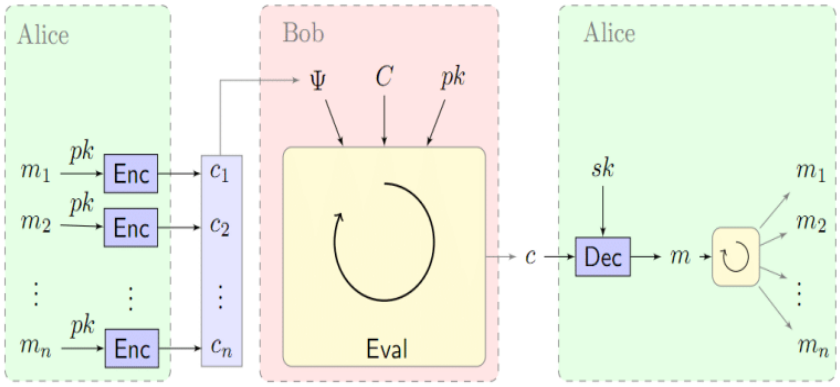
By employing these cryptographic primitives and techniques, the system provides a secure environment for managing and auditing medical data, ensuring that sensitive information remains confidential and that only authorized users have access to the data. The combination of encryption schemes and key management approaches contributes to a robust security infrastructure that supports the requirements of a medical audit system.

C. ADVANCED CRYPTOGRAPHIC SCHEMES FOR ENHANCED SECURITY

To improve the security properties of the medical audit system, various cryptographic schemes can be explored and potentially integrated into the system. Some of these schemes include:

1. **Homomorphic Encryption:**

Homomorphic encryption is a cryptographic technique that allows computations on encrypted data without decrypting it first. This means that encrypted data remains secure and confidential while still allowing for computations to be performed on it.



In medical audits, homomorphic encryption can be used to perform aggregate calculations on encrypted medical records without revealing the underlying data. This means that an auditor could calculate statistical values such as mean, median, mode, and standard deviation without having access to the individual records. **For example, the auditor could calculate the average blood pressure of a group of patients without accessing individual records. This preserves the privacy of patients while still providing valuable insights.**

Homomorphic encryption works by encrypting data using a mathematical function that allows for computations on the encrypted data. There are different types of homomorphic encryption, such as fully homomorphic encryption (FHE), partially homomorphic encryption (PHE), and somewhat homomorphic encryption (SHE). Each type has its own strengths and weaknesses.

FHE is the most powerful type of homomorphic encryption because it allows for any computation of encrypted data. However, it is also computationally intensive and impractical for large-scale use.

	Actions	Number of Operations
Partially Homomorphic Encryption (PHE)	One (Addition or multiplication)	Unlimited
Somewhat Homomorphic Encryption (SHE)	Two (Addition and multiplication)	Limited
Fully Homomorphic Encryption (FHE)	Two (Addition and multiplication)	Unlimited

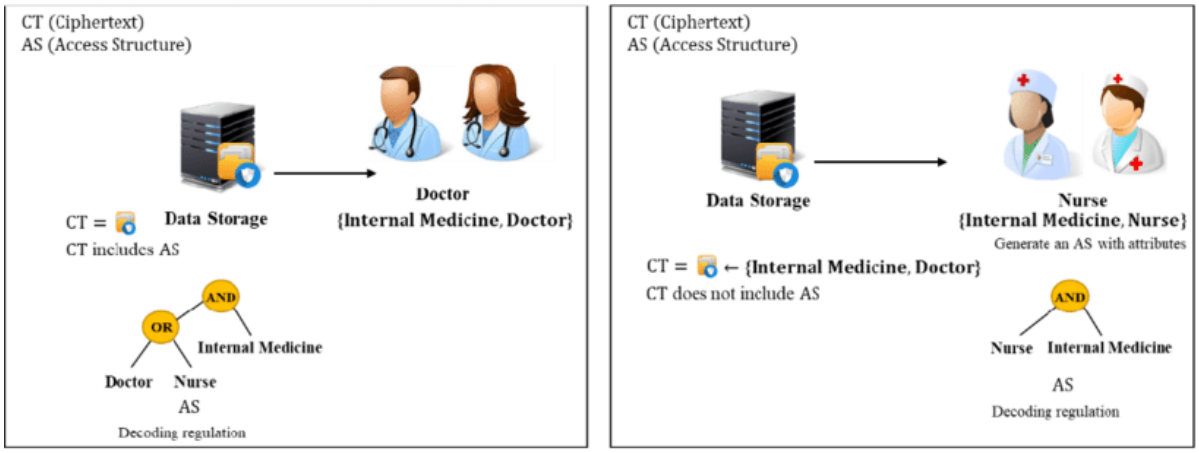
PHE and SHE, on the other hand, are more practical for real-world use. PHE allows only one type of computation on encrypted data, such as addition or multiplication. SHE allows for limited computations on encrypted data, such as addition, subtraction, and multiplication.

To use homomorphic encryption in a medical audit system, the data must first be encrypted using a homomorphic encryption algorithm. The auditor can then perform the necessary computations on the encrypted data without having access to the underlying data. The results of the computations can be decrypted using a decryption key.

Overall, homomorphic encryption provides a valuable tool for preserving privacy while allowing for useful computations of sensitive data, such as medical records. While there are still some practical limitations to its use, ongoing research is making progress in improving the performance and scalability of homomorphic encryption.

Reference: Vanin FNdS, Policarpo LM, Righi RdR, Heck SM, da Silva VF, Goldim J, da Costa CA. A Blockchain-Based End-to-End Data Protection Model for Personal Health Records Sharing: A Fully Homomorphic Encryption Approach. Sensors. 2023; 23(1):14. https://doi.org/10.3390/s23010014

2. **Attribute-based Encryption:**



Attribute-based encryption (ABE) is a type of encryption that enables access control based on user attributes. In traditional encryption schemes, access control is based on the identity of the user or the group they belong to. For example, a user may be granted access to certain files or resources based on their username or membership in a particular group.

In contrast, ABE allows access control based on a more fine-grained set of attributes. An attribute can be any information that describes the user, such as their job title or security clearance level. **For example, in a medical audit system, "doctor" attribute users could access sensitive patient records, while "nurse" attribute users may be limited to less sensitive records.**

The use of ABE in a medical audit system can provide several benefits. First, it can enable more granular access control, which can enhance privacy and security by ensuring that only authorized personnel can access certain types of information. This can help prevent data breaches or other unauthorized disclosures of sensitive information.

Second, ABE can help ensure access control policies are enforced consistently and fairly across the system. By basing access control on attributes rather than the user's identity, ABE can help prevent bias or discrimination based on gender, race, or nationality.

Finally, ABE can enable more flexible access control policies that can be easily adjusted or updated. For example, if a user's job title or security clearance level changes, their access to certain records can be updated automatically based on their new attribute values.

To put it simply, Attribute-based encryption (ABE) is a useful method for controlling access to information based on specific user attributes. Its application in a medical audit system can provide increased protection for sensitive data, ensure impartiality in access control, and allow for adaptable policies that can be changed.

Reference: Hwang Y-W, Lee I-Y. A Study on CP-ABE-Based Medical Data Sharing System with Key Abuse Prevention and Verifiable Outsourcing in the IoMT Environment. Sensors. 2020; 20(17):4934. https://doi.org/10.3390/s20174934

In summary, incorporating advanced cryptographic techniques such as homomorphic encryption and attribute-based encryption can significantly improve the security properties of the medical audit system. These techniques enable privacy-preserving computations, and fine-grained access control, which can all enhance security and privacy for patients, medical professionals, and auditors.

IV. HOW SYSTEM MEETS THE OUTLINED REQUIREMENTS

The system meets the outlined requirements in the following ways:

1. **Privacy:**

   The system ensures data privacy through cryptographic techniques such as RSA and AES-GCM. This guarantees that any server and client communication is encrypted and secure. User credentials are stored in a hashed format, protecting against unauthorized access.

2. **Identification and Authorization:**

   The server handles user authentication by verifying usernames and passwords. When a client connects to the server, it sends its username and password, which the server checks against its database. If the user is not registered, the server prompts the client to enter a password and user type (patient or auditor) for registration. This process ensures that only authorized users can access and modify data.

3. **Queries:**

   The system provides the ability to query user logs based on user type and patient ID. This is done through the query_user_logs() method in the Blockchain class. Patients can view their logs, while auditors can view logs for any patient. This ensures that users can only access the information they are authorized to view.

4. **Immutability:**

   The system ensures data immutability using blockchain implementation. The blockchain consists of a series of blocks containing a timestamp, patient ID, user ID, action, and hash information. The Block class has a calculate_hash() method that generates a unique hash for each block based on its contents and the previous block's hash. This creates a tamper-proof chain of records, as any modification to a block's contents would invalidate the subsequent blocks' hashes. The check_integrity() method in the Blockchain class verifies the integrity of the blockchain by ensuring that each block's hash matches its calculated hash and that its previous hash matches the hash of the previous block in the chain.

5. **Decentralization:**

   Although the system doesn't have a built-in decentralized structure, it has the potential to support one. The BlockChainFile class manages to store and access blockchain in a file, which could be expanded to permit several server instances to synchronize their blockchain copies. This would create a decentralized server network that collaborates to ensure the medical records security.

In conclusion, the secure server-client communication system meets the requirements of privacy, identification, authorization, queries, and immutability and provides the foundation for decentralization. By employing cryptographic techniques and blockchain implementation, the system ensures the security and integrity of medical audit data while allowing for proper access control and query functionality.

V. SYSTEM ASSUMPTIONS AND LIMITATIONS

A. SYSTEM ASSUMPTIONS

1. The code assumes that all clients and the server are trusted entities and will not attempt to exploit vulnerabilities or compromise the system's security.

2. The code assumes that the MedicalDataSystem and its associated methods, such as create_record, delete_record, update_record, etc., are implemented correctly and securely.

3. The code assumes that the underlying blockchain implementation and the BlockChainFile class handle the storage and retrieval of the blockchain securely and without data loss.

4. The code assumes that the cryptographic libraries, such as RSA and AES-GCM, are secure and implemented correctly and that potential vulnerabilities have been addressed.

5. The code assumes that all client-side user inputs are valid and free from malicious input that could lead to security vulnerabilities.

B. SYSTEM LIMITATIONS

1. Key Management:

The code does not handle the safekeeping and protection of cryptographic keys like RSA and AES-GCM. This could lead to a security breach if an unauthorized user gains access to them.

2. Denial of Service (DoS):

The system does not have any safeguards against DoS attacks. An attacker could flood the server with requests, making it unavailable to legitimate users.

3. Authentication and Authorization:

While the system implements basic user authentication, it does not address more advanced authentication and authorization mechanisms, such as multi-factor authentication or role-based access control.

4. Logging and Monitoring:

The system does not include comprehensive logging and monitoring mechanisms to detect and respond to security incidents or malicious activities.

5. Secure Communication:

Although the code employs cryptographic techniques for secure data transfer, it does not address other aspects of secure communication, such as ensuring data confidentiality, integrity, and availability during transmission.

6. **Scalability:**

   The system may need to scale better with many users and records, as it does not appear to use a distributed database or implement load-balancing mechanisms.

7. **Physical Security:**

   The code does not address the server's physical security or the devices clients use to access the system. Physical access to these devices could compromise the system's security.

C. SECURITY CHALLENGES NOT ADDRESSED

1. **Data Privacy and Compliance:**

   The code does not address data privacy regulations or compliance requirements like HIPAA, GDPR, or other regional data protection laws.

2. **Data Backup and Recovery:**

   The system does not have a data backup and recovery mechanism to protect against data loss due to hardware failures or other issues.

3. **Secure Software Development Lifecycle (SDLC):**

   The code does not provide information on whether it follows secure SDLC, such as incorporating security into all development process phases, including design, implementation, testing, and deployment.

4. **Patching and Vulnerability Management:**

   The code does not address how to handle software updates or vulnerability management, including identifying and remedying system security vulnerabilities or dependencies.

5. **Insider Threats:**

   The system does not have mechanisms to protect against insider threats, such as malicious employees or other authorized users who may attempt to compromise the system from within.

VI. IMPLEMENTATION

A. IMPLEMENTATION DESIGN

Implementing the secure server-client communication system for the medical audit use case follows a modular design, with separate components handling different aspects of the system. The code is divided into several files, each responsible for specific functionality. The following sections describe the design and implementation of each component in detail.

1. **Server.py:**

The server.py file handles user authentication, data integrity, and network communication. It relies on many cryptographic libraries for secure data transfer and leverages encryption, hashing, and padding techniques to protect sensitive information. The server is designed to efficiently handle multiple client connections, process client requests, and maintain high security.

The design of the server.py file is centered around the main function, which sets up the server socket, spawns a data integrity check thread, and enters a loop to accept and process incoming client connections. The server employs symmetric and asymmetric encryption, using RSA for initial key exchange and AES-GCM for subsequent communication. This design choice ensures both security and efficiency in encrypting and decrypting data.

2. **Client.py:**

The client.py program is responsible for securely connecting to the server and enabling the user to interact with the system. It manages sockets, verifies user identity, checks data integrity, and communicates with the server through encrypted messages.

The client establishes a secure connection with the server by exchanging encryption keys and setting up an AES-GCM cipher for future communication. After successful authentication, the client runs a continuous loop for interacting with the server, processing server responses, and allowing the user to access or modify data as needed.

3. **AuditClient.py:**

AuditClient.py contains the AuditClient class, which manages user interaction on the client side. The class takes a user type (either "patient" or "auditor") and a username as input and provides a user-friendly interface to interact with the system.

The design of the AuditClient class is centered around two main methods, 'printresponse' and 'ask', which handle displaying server responses and collecting user input, respectively. The class is designed to provide a simple and intuitive way for users to access the system's functionality based on their user type.

4. **AuditServer.py:**

The AuditServer.py file contains the AuditServer class, which is responsible for processing client requests on the server side. The class is designed to interact with a MedicalDataSystem for managing medical records and ensuring data integrity.

The primary method in this class, 'receive_data', receives a client request as a dictionary, processes it by calling the appropriate method on the MedicalDataSystem, and returns the result as a byte-encoded JSON string. The implementation of the MedicalDataSystem will have a significant impact on the functionality and performance of the system.

5. **MedicalDataSystem.py:**

The MedicalDataSystem.py file contains the MedicalData class, which manages medical records and interacts with the blockchain to maintain data integrity. The class is designed to provide a variety of methods for creating, deleting, updating, and retrieving medical records, as well as querying user logs and checking the integrity of the blockchain.

The MedicalData class is designed to work closely with the BlockChain class, which implements the underlying blockchain system. This integration ensures that all operations on medical records are securely logged and that data integrity is maintained throughout the system.

6. **BlockChain.py:**

   The BlockChain.py file provides the implementation of the blockchain system, which is used to maintain the integrity of the medical records within the medical audit system. The code is divided into three classes - Block, Blockchain, and BlockChainFile - each with its specific functionality and purpose.

   The Block class represents individual blocks in the blockchain, containing relevant data such as timestamps, user and patient IDs, actions performed, and hash values. The primary method in this class, 'calculate_hash', is responsible for generating the hash of a block based on its content. This design ensures that tampering with the block data will result in an invalid hash, allowing the system to detect data integrity issues.

   The Blockchain class represents the entire blockchain, consisting of a list of Block instances. The class is designed to provide methods for initializing the blockchain, querying user logs, logging operations, and checking the chain's integrity. The blockchain's design is focused on maintaining a secure and tamper-proof record of all medical data operations within the system.

   The BlockChainFile class handles the storage and retrieval of the blockchain to and from a file. The class is designed to provide methods for storing the current blockchain state in a file and loading the blockchain from a file. This design choice ensures that the blockchain data persists across sessions and can be audited or inspected anytime.

In conclusion, the secure server-client communication system for the medical audit use case follows a modular design, with each component handling specific aspects of the system. The code is organized into several files, each responsible for a particular functionality, resulting in a maintainable and extensible codebase. The system is designed to provide high security and data integrity, employing various cryptographic techniques and blockchain implementation to protect sensitive medical data.

B. EXECUTION INSTRUCTIONS

To execute the system on a Mac OS operating system using the command line interface (CLI), follow these steps:

1. First, create a Python virtual environment in the workspace by running the following command:
   - ➢ python3 -m venv myenv
2. Activate the virtual environment:
   - ➢ source myenv/bin/activate
3. Install the required packages using the requirements.txt file:
   - ➢ pip3 install -r requirements.txt
4. Rename the user.json, medical_data.json, and blockchain.txt files per your requirements.
5. Start the server by running the following command:
   - ➢ python3 server.py
6. In a separate terminal, start the client by running the following command:
   - ➢ python3 client.py

On the client side:

1. Enter the username.
2. Enter the password.
3. If you are a new or unregistered user, enter the user type (either "patient" or "auditor") by pressing Y for patient or any other input for auditor

Once these steps are completed, you will have successfully entered the system.

After successfully entering the system, you will be presented with a menu of options based on your user type (patient or auditor). The options allow you to interact with the medical audit system to perform various tasks. Here's what you can expect:

Patient	Auditor
1. Create a new medical record.	1. Create a new medical record of any patient
2. Update an existing medical record.	2. Update an existing medical record of any patient.
3. Delete a medical record.	3. Delete a medical record of any patient
4. Retrieve and print medical records.	4. Retrieve and print medical records of any patient
5. Copy a medical record.	5. Copy a medical record.
6. Query your own logs from the blockchain.	6. Query logs from the blockchain for any patient.

To perform these tasks, enter the corresponding number from the menu and follow the on-screen prompts. Depending on the selected option, you may need to provide additional information, such as the patient ID or the medical record details like temperature and blood pressure.

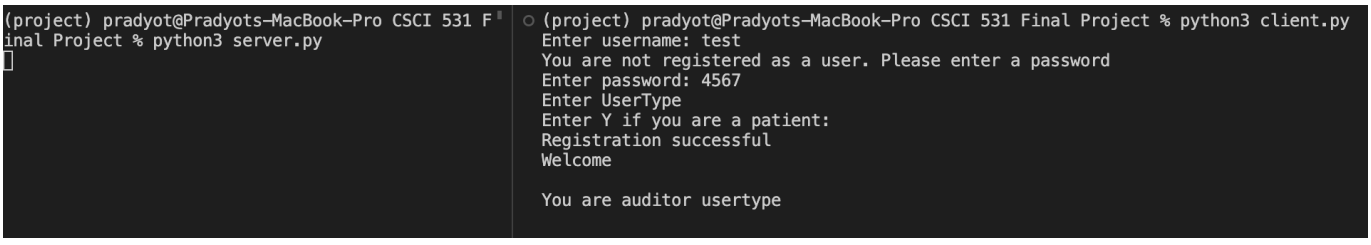
You can continue interacting with the system and performing various operations until you exit the system by not pressing "Y" for the continue option. Once you exit, the client will disconnect from the server, and you can close the terminals.

In the server side after an interaction with a client has ended you can end the server by pressing "Q" or continue with connecting to another client by pressing anything else.

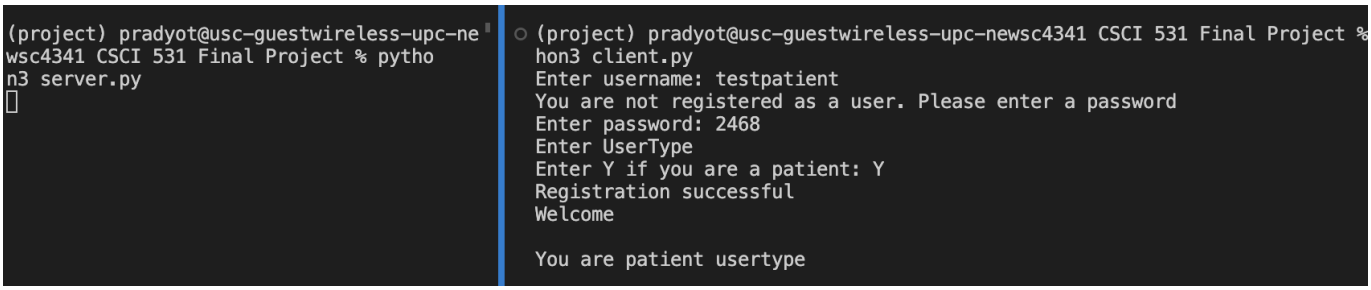
C. SCREENSHOTS

1. User Creation/Login:

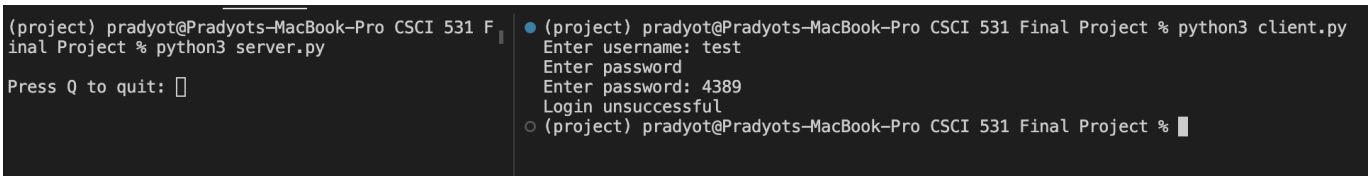
User Creation (auditor):



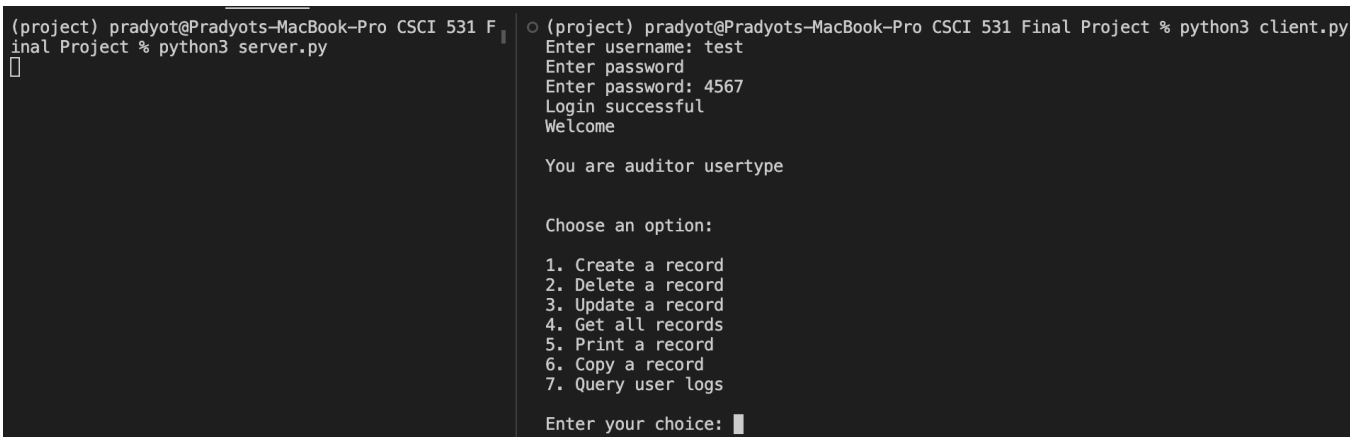
User Creation (patient):



User Login (unsuccessful):

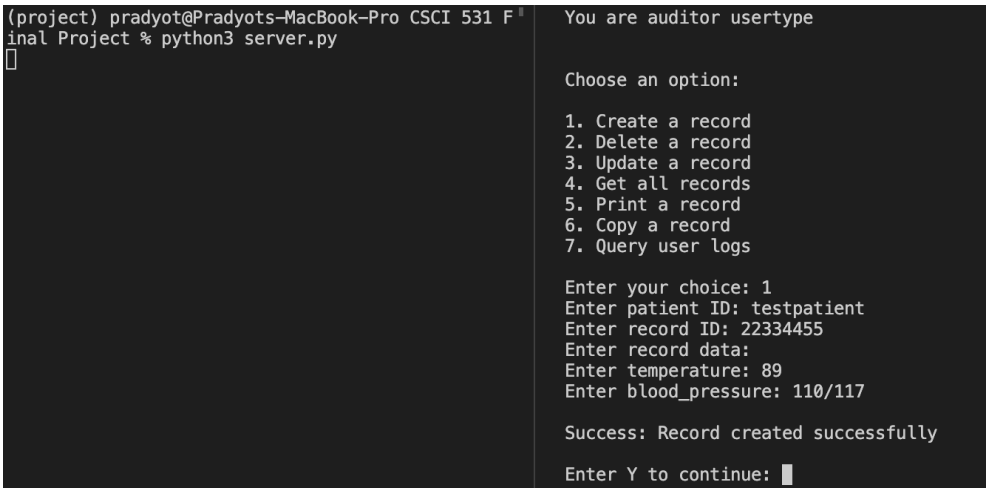


User Login (successful):

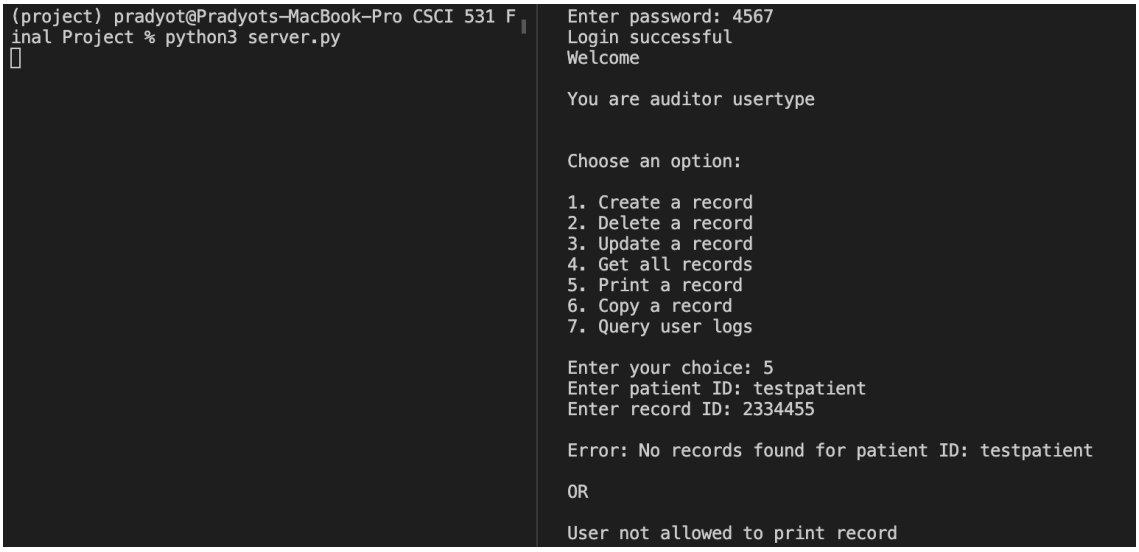


2. Record Creation/Updation and Printing:

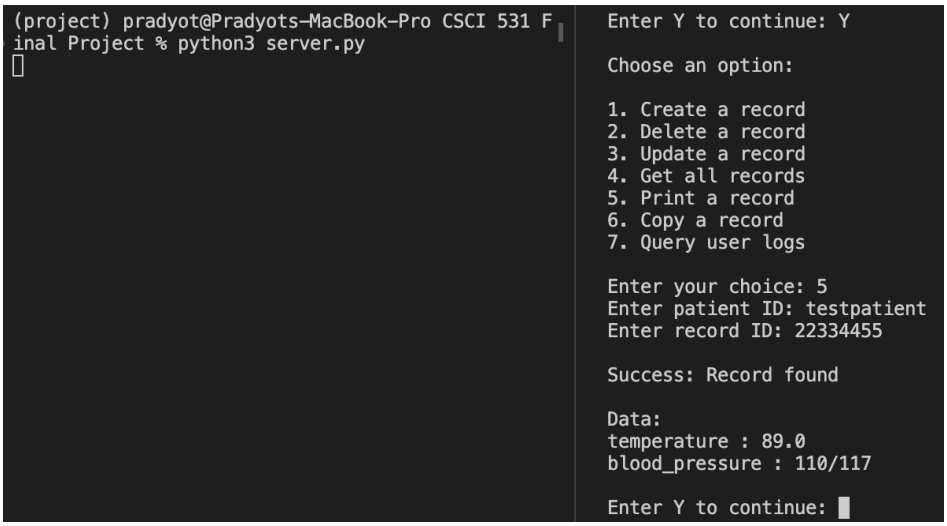
Record Creation (auditor):



Record Print (auditor) (unsuccessful):



Record Print:



Record Updation (auditor):

```
(project) pradyot@Pradyots-MacBook-Pro CSCI 531 Final Project % python3 server.py
```

```
Enter Y to continue: Y

Choose an option:

1. Create a record
2. Delete a record
3. Update a record
4. Get all records
5. Print a record
6. Copy a record
7. Query user logs

Enter your choice: 3
Enter patient ID: testpatient
Enter record ID: 22334455
Enter record data:
Enter temperature: 100
Enter blood_pressure: 120/121

Success: Record updated successfully

Enter Y to continue:
```

Record Print after Updation:

```
(project) pradyot@Pradyots-MacBook-Pro CSCI 531 Final Project % python3 server.py
```

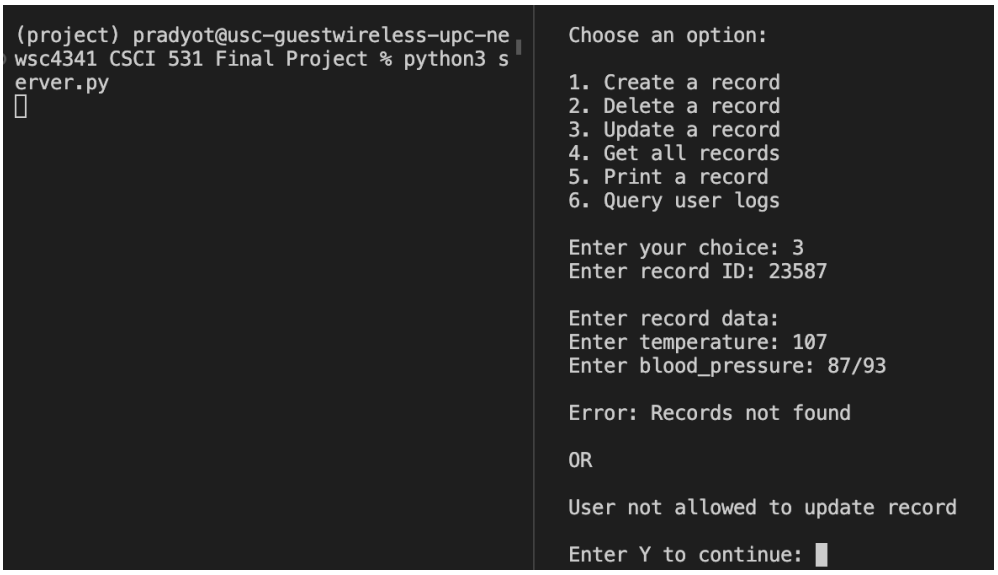
```
Enter Y to continue: Y

Choose an option:

1. Create a record
2. Delete a record
3. Update a record
4. Get all records
5. Print a record
6. Copy a record
7. Query user logs

Enter your choice: 5
Enter patient ID: testpatient
Enter record ID: 22334455

Success: Record found

Data:
temperature : 100.0
blood_pressure : 120/121

Enter Y to continue:
```

Record Creation (patient):

```
(project) pradyot@usc-guestwireless-upc-newsc4341 CSCI 531 Final Project % python3 server.py
```

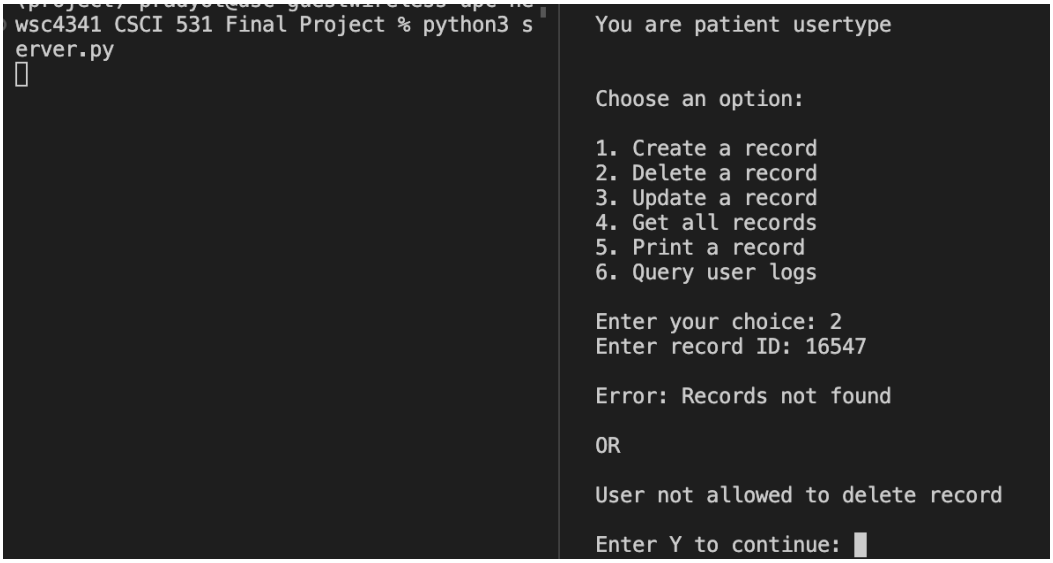
```
You are patient usertype


Choose an option:

1. Create a record
2. Delete a record
3. Update a record
4. Get all records
5. Print a record
6. Query user logs

Enter your choice: 1
Enter record ID: 13579

Enter record data:
Enter temperature: 104
Enter blood_pressure: 88/98

Success: Record created successfully

Enter Y to continue:
```

Updating other users record (patient) (unsuccessfull):



3. Delete a record:

Record Deletion (patient)(unsuccessfull):



Record Deletion (auditor):

```
wsc4341 CSCI 531 Final Project % python3 s
erver.py
```

```
Welcome

You are auditor usertype


Choose an option:

1. Create a record
2. Delete a record
3. Update a record
4. Get all records
5. Print a record
6. Copy a record
7. Query user logs

Enter your choice: 2
Enter patient ID: testpatient
Enter record ID: 13579

Success: Record deleted successfully

Enter Y to continue:
```

4. **Query logs:**

Query logs (patient):

```
(project) pradyot@usc-guestwireless-upc-ne
wsc4341 CSCI 531 Final Project % python3 s
erver.py
```

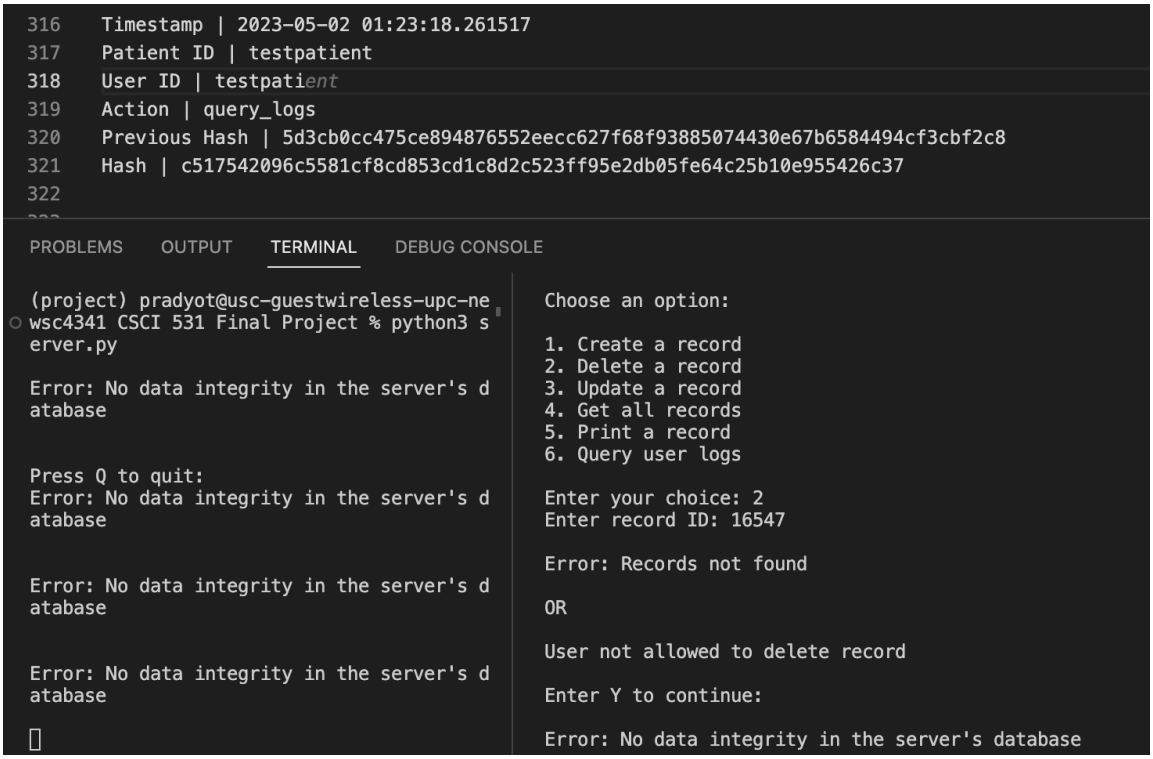
```
3. Update a record
4. Get all records
5. Print a record
6. Query user logs

Enter your choice: 6

Success: Logs found for user testpatient

Data:
Logs

Datatime               User ID         Patient ID              Action
-------------------------------------------------------------------------
2023-05-02 00:04:54      test           testpatient                    create
2023-05-02 00:24:05      test           testpatient                    print_record
2023-05-02 00:25:19      test           testpatient                    update
2023-05-02 00:27:53      test           testpatient                    print_record
2023-05-02 01:13:27      testpatient            testpatient                    create
2023-05-02 01:17:44      testpatient            testpatient                    update
2023-05-02 01:23:18      testpatient            testpatient                    query_logs
```

Query logs (auditor):

```
(project) pradyot@usc-guestwireless-upc-ne
wsc4341 CSCI 531 Final Project % python3 s
erver.py
```

```
7. Query user logs

Enter your choice: 7
Enter patient ID: testpatient

Success: Logs found for user testpatient

Data:
Logs

Datatime               User ID         Patient ID              Action
-------------------------------------------------------------------------
2023-05-02 00:04:54      test           testpatient                    create
2023-05-02 00:24:05      test           testpatient                    print_record
2023-05-02 00:25:19      test           testpatient                    update
2023-05-02 00:27:53      test           testpatient                    print_record
2023-05-02 01:13:27      testpatient            testpatient                    create
2023-05-02 01:17:44      testpatient            testpatient                    update
2023-05-02 01:23:18      testpatient            testpatient                    query_logs
2023-05-02 01:34:36      test           testpatient                    delete
2023-05-02 01:35:26      test           testpatient                    query_logs
```

5. **Integrity Error Message:**



VII. DEMO RECORDING

CSCI 531 Final Project Demo.mp4

REFERENCES

1. Fernández-Alemán, J. L., Señor, I. C., Lozoya, P. Á., & Toval, A. (2013). Security and privacy in electronic health records: a systematic literature review. Journal of biomedical informatics, 46(3), 541–562. https://doi.org/10.1016/j.jbi.2012.12.003

2. Kuo, T. T., Kim, H. E., & Ohno-Machado, L. (2017). Blockchain distributed ledger technologies for biomedical and health care applications. Journal of the American Medical Informatics Association : JAMIA, 24(6), 1211–1220. https://doi.org/10.1093/jamia/ocx068

3. A. Azaria, A. Ekblaw, T. Vieira and A. Lippman, "MedRec: Using Blockchain for Medical Data Access and Permission Management," 2016 2nd International Conference on Open and Big Data (OBD), Vienna, Austria, 2016, pp. 25-30, https://doi.org/10.1109/OBD.2016.11.