FUTURE WORK

(**Multiple patients-multiple doctors**)

SECTION 3: SYSTEM DESIGN

The design of the entire system has taken the following two design paradigms into account:

* First, design a simpler problem, and then use the insight to solve the more complex problem.

The simpler problem we handled first was to design a *simple* e-health records distributed management system that allowed *one patient* to share their e-health records with *only* *one doctor*. Once we were able to design this simple system, we moved in and designed more complex systems listed below:

* A e-health records distributed management system to allow *one patient* to share their e-health records with *multiple doctors*.
* A e-health records distributed management system to allow *multiple patients* to share their e-health records with *multiple doctors.*
* *An end-to-end encryption* health e-record distributed management system to allow *multiple patients* to *securely share* their e-health records with *multiple doctors.*

This approach of taking small, incremental steps and progressively building upon them has helped us significantly to focus on and improve each core functionality in the system and then easily integrate them into one large system.

* Use analogies to derive techniques for the problem at hand from solutions to a different problem. The problem at hand was how to allow a patient to *securely* *share* their e-health records with multiple doctors without having the patient to individually encrypt their e-health records for each doctor using the doctor’s public key.

The analogy used was the following:

For some secure messaging systems with users holding a key pair and making their public keys online accessible, a *user A* can send a secure message to a *selected* group of users using "hybrid encryption and decryption” in the following *way*:

* The user A encrypts their message using their symmetric secret key and then encrypts their symmetric secret key using the public key of each recipient of the message.
* The recipient of the message then uses their private key to decrypt the symmetrically encrypted secret key of the sender and then uses the secret key to decrypt the symmetrically encrypted message.

This same way can be applied to patients who would like to securely share their e-health records with multiple doctors. In this case, e-health records of patients are to be encrypted symmetrically using the patient's secret key before being uploaded to IPFS. When a patient would like to grant a health care provider access to their health records, the patient encrypts their secret key asymmetrically using the public key of the health care provider (i.e. doctor), and the patients public key if necessary. The health care provider can decrypt the patient's secret key using their private key and use the secret key of the patient to decrypt the patient's health records when they retrieve it from IPFS.

**3.1 System-level design overview**

The Blockchain-based distributed secure e-health record management system has the following system-level design:

A close up of a device

Description automatically generated

Figure1 : High level use case

As shown in the figure above, the system relies on IPFS for decentralized file storage and data integrity, Ethereum blockchain for IPFS file hash storage, Infura for API access to the Ethereum network and IPFS, end-to-end encryption for privacy protection, and a decentralized app (Dapp) for communication with both the Ethereum blockchain and IPFS using Web3.JS, Metamask, and React JS.

The table below summarizes the types of operations each of the patient and the doctor can perform:

Table 1: Type of operations the Dapp users can perform

|  |  |  |
| --- | --- | --- |
| Type of operation | Patient | Doctor |
| register as a user | Yes | Yes |
| Upload e-health records | Yes | No |
| Access e-health records | Yes | Yes |
| Grant access | Yes | No |
| Revoke access | Yes | No |

The following diagram shows the process of user registration for the Dapp:

A close up of a device

Description automatically generated

Figure 2: User registration

As shown above, this process relies on three types of interactions: user-to-Dapp interaction,

Dapp-to-IPFS interaction, and Dapp-to-blockchain interaction. It is worth noticing that the doctor does provide *an encrypted version of their private key* for privacy protection and for later

use of this encrypted private key for decrypting *the encrypted version of the secret key* of the patient.

Please note that the patient also provides an encrypted version of their private key, which is not used in the current implementation of the system. It was only added so that we can use it when we extend our system in the future to allow patients to securely exchanges messages with their doctors using their public and private keys.

The following diagram shows in detail the types of operations a patient can perform:

A close up of a map

Description automatically generated

Figure 3: Patient operations

The above diagram also shows three types of interactions: user-to-Dapp interaction,

Dapp-to-IPFS interaction, and Dapp-to-blockchain interaction.

It is worth mentioning that the patient can still grant a doctor an access to their e-health records (i.e. files) after they have revoked the doctor’s access. Also, the Dapp performs symmetric encryption and decryption to files using the secret key of the patient, asymmetric encryption to the secret key of the patient using the public key of the access-approved doctor, as well as provides the patients with two options for file access: view or download the file.

The following diagram shows in detail the types of operations a doctor can perform:

A picture containing screenshot

Description automatically generated

Figure 4: Doctor operations

The above diagram shows four types of interactions: user-to-Dapp interaction,

Dapp-to-IPFS interaction, Dapp-to-blockchain interaction, and Dapp-to-Dapp interaction (i.e. Dapp *internal* processing).

It is worth noticing that the Dapp here dominantly handles the processing. The Dapp fetches all the patients’ and the doctors’ related files and encrypted keys from both the IPFS and the Ethereum blockchain and performs symmetric and asymmetric decryption before making the files of the patients available for access-approved doctor for view or download.

**3.2 Design decisions**

|  |  |  |
| --- | --- | --- |
| # | Design decision | Discussion |
| 1 | IPFS for data storage | * peer-to-peer file system​. * content-addressed data​. * tamper-proof​. * data is public, therefore needs to be encrypted.   With IPFS, data is content addressed (i.e. you tell the network *what* to look for, and the network figures out where to look), rather than location-addressed (you tell the network *where* to look and the network sends back what it found).​  Because data is content addressed in IPFS, this provides two advantages: ​   * First, it makes the network more resilient. The content with a hash could be stored on multiple nodes. if one node that is storing that content goes down, the network will just look for the content on another node. This is very important for our system as it ensures that the patient’s data is available and that doctors can access the data when needed​. * Second, IPFS provides a built-in way of determining whether the content of a file has been tampered with or not. For example, a user can determine whether a file they retrieved from IPFS had been tampered with or not by rehashing the retrieved file and comparing the hash result with the hash they used to retrieve the file from IPFS. If the two hashes are different, then the user knows that the content of the file was tampered with. Tamper-proof feature is important for our Dapp. [1]. |
| 2 | Hybrid encryption  for privacy protection and secret key sharing | In order to provide privacy protection and share the secret key with more than one entity:   * Files are *symmetrically* encrypted for performance reasons. ​ * The secret key for the *symmetric encryption* above is *asymmetrically* encrypted for *each* doctor. * The secret key for the *symmetric decryption* is asymmetrically decrypted by each doctor.   Most symmetric encryption and decryption systems are much faster than most asymmetric ones. Encrypting the symmetric key with the expensive asymmetric algorithm and then using that symmetric key for decryption has much better performance, even when there is only one doctor [2]. ​ |

**SECTION 4: IMPLEMENTATION**

The next sub-sections include the npm configurations, truffle setup, smart contract compilation, smart contract migration, MetaMask wallet manager setup, and the operations when a user (i.e. a patient or a doctor) using the Dapp.

**4.1 npm configurations**

* Installing npm

npm install

* Installing IPFS

npm install --save ipfs-http-client

npm install --save ipfs-api

* Installing Crypto-js

npm i crypto-js

npm install crypto-js

* Installing pubkey/eth-crypto

Npm install eth-crypto --save

**4.2 Truffle Configuration**

* Creating a Truffle Project

The **truffle init**command creates a truffle project with the following structure:

* contracts/: stores the smart contract.
* migrations/: instructions for deploying the smart contract in the “contracts” folder.
* test/: JavaScript tests for the smart contract
* truffle-config.js: a JavaScript file used for describing how the smart contract code is deployed and is shown in the figure below.

**A screenshot of a computer

Description automatically generated**

**Figure 4: Truffle Configuration**

Also, a local private blockchain, Ganache-GUI, is used to deploy the smart contract to. TheGanache-GUI provides 10 accounts that can be imported to MetaMask.

* Smart contract compilation and migration and React server

The smart contract is deployed using the truffle command “truffle compile”, as shown in the figure below.

**A screenshot of a computer screen

Description automatically generated**

Figure 5 : Smart contract compilation and migration and React sever.

In order to deploy the smart contract to Ganache, truffle needed to be configured again. In addition to the first provided migration file, which tells truffle to first deploy the migrations contract, a second migration file was added. This added file is the second file to be run in the migration process, as shown above in the figure.

Also, the command “npm start” was used to start the React Server and display the main page of the Dapp.

**4.3 MetaMask setup**

MetaMask has been connected to Ganache before importing the second, third, fourth, and fifth Ganache accounts to MetaMask, as shown below in the figure.

**A screenshot of a computer

Description automatically generated**

Figure 6: MetaMask setup

**4..4 User registration**

Each Ganache-GUI account has a public key and a private key. The private key of an account can be easily found by clicking on the key logo next to the account address. To obtain the corresponding public key of an account, we needed to use the *eth-crypto library* in JavaScript to obtain the corresponding the public key from the private key.

A table summarizing the key pair for each account used as well as the script used to obtain the public key is shown in the figure below.

A screenshot of a computer

Description automatically generated

Figure 7: Key pair for MetaMask accounts

* Patient 1 registration

Patient 1 registers in the Dapp by entering their account address, followed by their encrypted version of their private key and their user type. Once the patient has registered successfully, the Dapp view changes from *registration view* to *patient view* in which the patient can upload files, view files, or grant an e-health record access to a doctor, as shown below.

A screenshot of a computer screen

Description automatically generated

Figure 8: Patient 1 registration

Also, closer look at the above figure reveals that the encrypted version of the patient’s private key has been stored successfully on IPFS. This choice of storing encrypted versions of private keys of the users on IPFS has been made to overcome limitation with the use of Web3.JS with Ganache. These limitations will be discussed in the next section, technical problems sections.

* Doctor 1 registration

Doctor 1 registers in the same way as a patient does, except that they select “Doctor” as their user type. Once the doctor has registered successfully, the Dapp view changes from the registration view to the doctor view, as shown below.

A screenshot of a computer screen

Description automatically generated

Figure 9: Doctor 1 registration

Also, a closer look at the above figure reveals that the encrypted version of the private key of the doctor has been stored successfully on IPFS.

* Patient 2 and Doctor 2 registration

There was no need to show the registration steps for Patient 2 and Doctor 2 since their registration steps are very similar to their counterparts.

**4.5 Interaction between patients and doctors**

The types of interactions between patients and doctors have already been defined in Table 1 above. Here, we show screenshots of those interactions.

* Patient 1 uploads a file and grants an e-health record access to Doctor 1

A screenshot of a computer screen

Description automatically generated

Figure 10: Patient 1 uploads a file and grants an access to Doctor 1

A closer look at the above figure reveals that the process of uploading an encrypted file to IPFS goes through the following:

* The file is taken as input.
* The file is converted to a buffer.
* The buffer is symmetrically *encrypted* using the symmetric key of the patient to obtain a *ciphertext*.
* The *ciphertext* is *buffered* and then uploaded to IPFS.
* The *hash* of the *buffered ciphertext* returned by IPFS is stored on the blockchain.

Also, the figure above shows that the patient can also view or download the uploaded file. To do so, the patient clicks on the “view file” button to retrieve the file form IPFS. The process of retrieving the file from IPFS is the exact reverse of the process of uploading the file to IPFS. Once the file has been retrieved and is in the form of a buffer, the Dapp passes the file to a blob object in order to make it available for the patient for view or download.

Furthermore, the above figure shows that the patient has granted access to Doctor 1 by entering the account address and public key of Doctor 1. The public key of Doctor 1 is needed for the Dapp to encrypt the secret key of Patient 1 so that Doctor 1 can access this secret key using their private key.

* Patient 2 uploads a file and grants an access to Doctor 1

Patient 2 follows the same way for uploading files and granting access to doctors as Patient 1 does. In this case, Patient 2 makes a file upload to IPFS and grant access to Doctor 1, as shown below.

A screenshot of a computer

Description automatically generated

Figure 11: Patient 2 uploads a file and grants an access to Doctor 1

* Doctor 1 now has access to the files of Patient 1 and Patient 2

A screenshot of a cell phone screen with text

Description automatically generated

Figure 12: Doctor 1has now access the files of Patient 1 and Patient 2

* Patient 2 grants an access to Doctor 2

A screenshot of a computer screen

Description automatically generated

Figure 13: Doctor 2 now has access to files of Patient 2

* Patient 2 uploads a second file, followed by Patient 1 revoking access from Doctor 1

A screenshot of a computer screen

Description automatically generated

Figure 14: Patient 2 uploads a second file, followed by Patient 1 revoking access from Doctor 1

As a result of this interaction, Doctor 1 now can only access to files of Patient 2 while Doctor 2 can now access the new file of Patient 2.

* Patient 1 re-grants file access to Doctor 1

A screenshot of a social media post

Description automatically generated

Figure 15: Patient 1 re-grants file access to Doctor 1

As a result of this interaction, Doctor 1 now has re-gained access to the files of Patient 1.

**SECTION 5: TECHNICAL PROBLEMS ENCOUNTERED**

Below is a discussion of the technical problem.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Problem | Solution | Discussion |
| 1 | Solidity does not support storing objects on the blockchain | Re-architect our solution. | When a patient grants an access to doctor, the Dapp encrypts the secret key of the patient using the public key of the access-granted doctor. The result of this encryption is an object with four elements, which cannot be stored directly on the blockchain as solidity does not support this feature.  To address this limitation, we had to the data structure *struct* in Solidity. We had to individually retrieve the four elements of the object in ReactJS and pass them to the *GrantAccessToDoctor* function in the Solidity smart contract to create struct that includes these four elements. |
| 2 | Solidity does not support returning an array of structs | Find some workaround | After we were able to address the previous limitation in Solidity, we then had to address another limitation in Solidity, which is solidity does not return an “array of structs”, as we expected.  Solidity returns an "array of arrays" instead. To find some workaround to resolve the issue, we needed to create an object named "encryptedPassphrase" and assign specific values from the returned array of arrays to the elements of the object, as shown in the app.js lines (74-81).  Solidity is still an evolving programming language that needs to include support for commonly used features in other programming languages in order to make it easy for developer to focus only on coding their solutions rather finding solutions to limitation to the Solidity programming language. |

REFERENCES:

[1] <https://blog.cloudflare.com/distributed-web-gateway/>

[2] <https://security.stackexchange.com/questions/110143/asymmetric-encryption-for-multiple-recipients>