

# CSU34031 Advanced Telecommunications

## Secure Cloud Storage - John Sinclair - 16325734

The repository for this project can be found [here](#) and a live deployment can be found at <https://secure-cloud-storage.vercel.app>

### Task

The aim of this project is to develop a secure cloud storage application for Dropbox, Box, Google Drive, Office365 etc. For example, your application will secure all files that are uploaded to the cloud, such that only people that are part of your “Secure Cloud Storage Group” will be able to decrypt uploaded files. Any member of the group should be able to upload encrypted files to the cloud service. To all other users the files will be encrypted.

You are required to design and implement a suitable **key management** system for your application which employs **public-key certificates** that allows users of the system to **share files securely**, and also allows one to **add or remove users** from the group. You are free to implement your application for a desktop or mobile platform and make use of any open source cryptographic libraries.

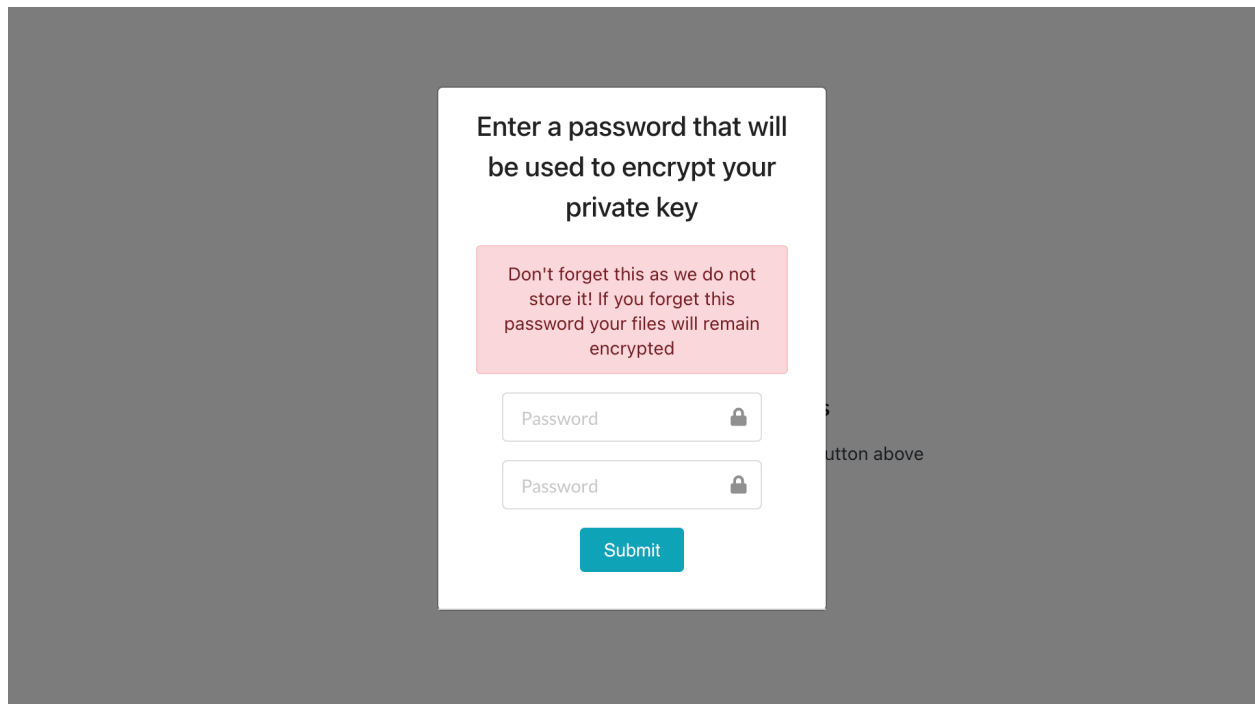
### Approach Overview

I developed my application using the JavaScript frontend framework ReactJS. The web applications backend is implemented using Firebase, using both the document oriented Firestore as well as Firebase Storage for the encrypted files themselves. I used both the crypto-js and hybrid-crypto-js libraries to implement the applications key management system for file encryption and decryption.

### Application Overview

I implemented my login using Google authentication, I chose this for its simplicity and universal appeal. When a user logs in, the application checks the database to see whether the user is registered with the service, if not the following prompt is shown. It is requesting a password from the user that will be used to encrypt the user's private key so as not to store a plain text version of it on the server. When the user has decided on a password the application generates a new RSA key pair and encrypts the RSA private key using AES encryption with the password as the secret key. Note, the password is not stored anywhere, if the user forgets it they will be unable to decrypt their private key and therefore, any files they had stored on the cloud service. While

this may seem an inconvenience to users, it ensures the utmost security as not even the server admin has the ability to decrypt and user files.



The screenshot shows a white modal form centered on a dark gray background. The form has a title, a warning box, two password input fields, and a submit button.

Enter a password that will be used to encrypt your private key

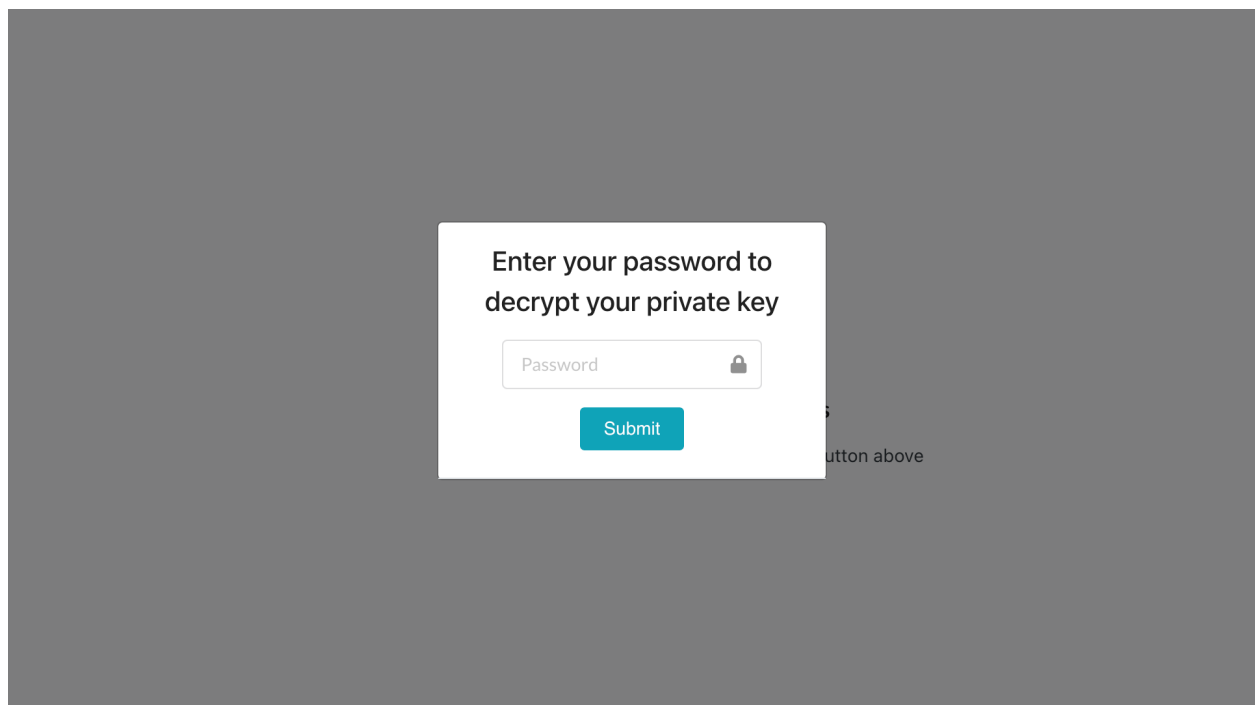
Don't forget this as we do not store it! If you forget this password your files will remain encrypted

Password

Password

Submit

Similarly, if a user is already registered with the service, they will be prompted to enter their password in order to decrypt their encrypted RSA private key stored on the server.



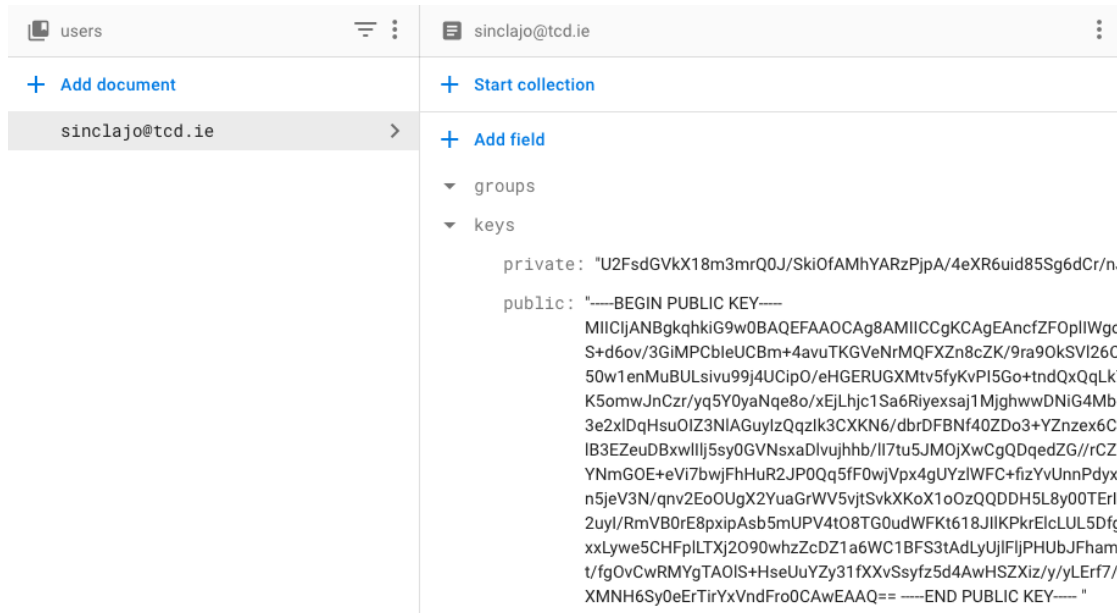
The screenshot shows a white modal form centered on a dark gray background. The form has a title, a password input field, and a submit button.

Enter your password to decrypt your private key

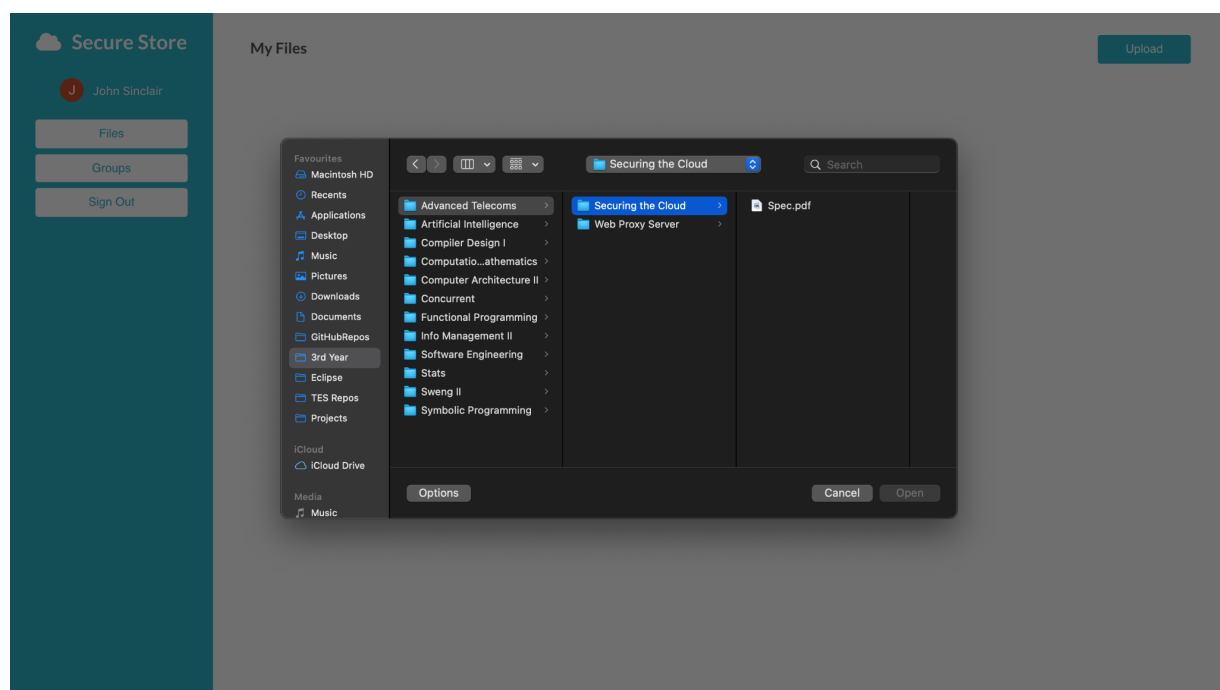
Password

Submit

This is a very effective mechanism as the user cannot access the application without first inputting the correct password, the private key decryption fails if the password is incorrect and the prompt is triggered again. The picture below is from the Firebase document database, as you can see the private key is entirely encrypted and completely useless without the password.



Once the user has logged in and inputted their password they have full access to the application. The service is split into two sections, personal files and the group files. Users can then upload files to their own or to a group by pressing the upload button in the top right corner. This opens a window for them to select the file for upload.



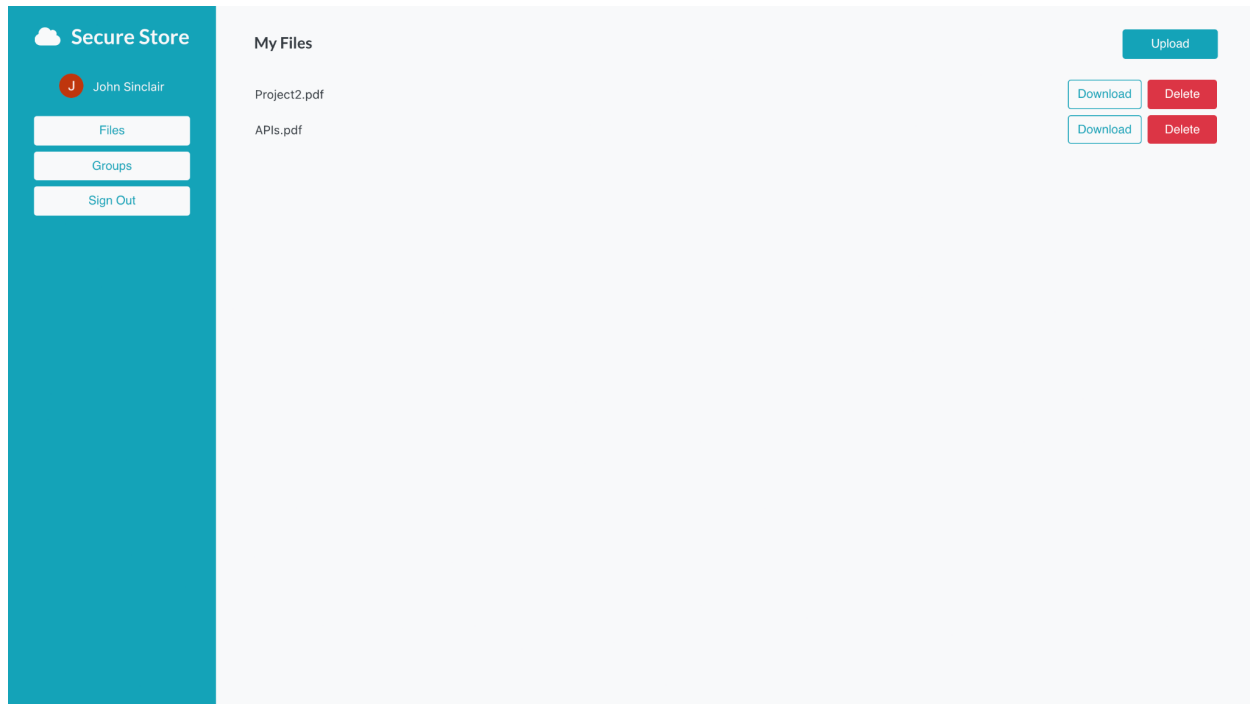
Once a file is selected, the file is read into the application by the `handleUpload` function. The method has a number of checks, such as ensuring a valid file has been selected, checking if the size is less than the limit (currently 50mb) and checking whether the file already exists in the cloud storage database. If the file is already stored the upload process is slightly different, updating the file in the cloud instead of adding a new copy, eliminating duplicates.

```
const isDuplicate = (passed) => {
  let flag = false;
  files.forEach((f) => {
    if (f.filename === email + "&^%" + passed) flag = f;
  });
  return flag;
};

const handleUpload = (e) => {
  setUploading(true);
  if (e.target.files.length > 0) {
    let file = e.target.files[0];
    if (file.size < maxfilesize * 1024 * 1024) {
      let dupe = isDuplicate(file.name);
      addUserFile(file, email, keys.public, dupe).then(() => {
        setFlag(!flag);
        setUploading(false);
      });
    } else {
      setUploading(false);
      setTooLarge(true);
      setTimeout(() => setTooLarge(false), 3000);
    }
  }
  e.target.value = null;
};
```

The `addUserFile` then uploads the file. The file is first converted to a data url, this is to allow for easy encryption, the resulting data url is encrypted and added to the cloud storage. Then, an accompanying document is added to the firestore database with owner information and a file download link.

```
export const addUserFile = async (file, email, key, dupe) => {
  return new Promise(async (resolve) => {
    if (dupe) await deleteFile(dupe);
    let filename = email + "&^%" + file.name;
    const storageRef = app.storage().ref();
    const fileRef = storageRef.child(filename);
    let dataurlfile = await fileToDataURL(file);
    let encdataurl = await encryptData(dataurlfile, key);
    fileRef.putString(encdataurl).then(() => {
      fileRef.getDownloadURL().then(async (url) => {
        files.add({
          owner: email,
          name: filename,
          url: url,
          group: false,
        });
        resolve();
      });
    });
  });
};
```



Uploaded files can be seen in the 'Files' tab, where users can download as well as delete any files currently stored. The `downloadFile` function handles the download process. It takes two arguments, the file (which is the file information document containing the download url), and the user's private key. First the file (in the form of an encrypted data url) is fetched using the download url, the resulting data url is then decrypted using the `decryptData` function, converted back into a JS File object then the browser download process starts. To the user this is a seamless process, they simply have to press the download button.

```
export const downloadFile = async (file, key) => {
  return new Promise(async (resolve) => {
    var reader = new FileReader();
    let data = await fetch(file.url);
    let blob = await data.blob();
    reader.readAsText(blob);
    reader.onload = async () => {
      let decdata = await decryptData(reader.result, key);
      let decfile = await dataUrlToFile(decdata.message);
      let fileurl = URL.createObjectURL(decfile);
      // window.open(fileurl);
      let a = document.createElement("a");
      a.href = fileurl;
      a.download = file.filename.split("&^%")[1];
      a.click();
      resolve();
    };
  });
};
```

Group functionality is implemented much the same as regular users are in the application. Any user can create a group, giving it a name and a password. The system then generates a set of RSA keys, again encrypting the private key with the password using AES encryption, the encrypted key pair is then added to the group object along with the name and admin email. The group object is then added to the database.

Note, this is as far as I got implementing the group functionality but the following is how I had intended to implement it: Within the group, the creator or admin of the group can then add people to the group via their email. If the email is registered as a user of the service the groups unique auto generated ID is added to the array of groups in the users document in the database. Then when the added user navigates to the 'Groups' tab on their device the new group will be displayed. In order for the added user to be able to access and decrypt the group's shared files they must enter the password set by the admin, decrypting the private key.

## Key Management System

I used a combination of two node libraries to implement the applications key management system for encryption and decryption, crypto-js and hybrid-crypto-js. The RSA encryption functionality was provided by hybrid-crypto-js and the AES encryption used to encrypt the RSA private key was provided by crypto-js. All encryption related functions can be found inside Crypto.js in the api folder. For security, all encryption and decryption takes place client side, the only info remotely stored is the users encrypted RSA key pair and their encrypted files in the form of an encrypted data url.

The user information that is stored remotely is in the form of a document as it is stored in a NoSQL Firestore database. For each document the users email, provided by google authentication is used as the documents id, the documents contained two fields, the first was an array of group ids that the user was a member of and the other the user's encrypted RSA key pair. These documents act as public key certificates in my system, directly associating the users email with ownership of the key pair. The Google login effectively adds another layer of security for prospective users.

```
import { Crypt, RSA } from "hybrid-crypto-js";
import CryptoJS from "crypto-js";

const crypt = new Crypt();
const rsa = new RSA();
const aes = CryptoJS.AES;
```

First the relevant components of each library were imported and assigned to a constant. The first function is the generateKeys function, it very simply creates then returns an RSA key pair.

```
export const generateKeys = () => {
  return new Promise((resolve) => {
    rsa.generateKeyPair((keyPair) => {
      resolve({
        public: keyPair.publicKey,
        private: keyPair.privateKey,
      });
    });
  });
};
```

Next, the application required functions to encrypt and decrypt the RSA private key using a user password.

```
export const decryptPrivateKey = async (pair, password) => {
  return new Promise((resolve) => {
    try {
      let encpriv = pair.private;
      let decrypted = aes.decrypt(encpriv, password);
      decrypted = decrypted.toString(CryptoJS.enc.Utf8);
      resolve({ public: pair.public, private: decrypted });
    } catch (e) {
      resolve(false);
    }
  });
};
```

The decrypt function is slightly longer as it contains error handling in the case the user incorrectly inputs their password. This way a user can repeatedly input an incorrect password without the application crashing.

The last two functions required were to encrypt and decrypt the files that were passed as parameters in the form of a data url. The data is encrypted using the user's public key.

```
export const encryptData = async (data, key) => {
  return new Promise((resolve) => {
    resolve(crypt.encrypt(key, data));
  });
};
```

```
export const decryptData = async (data, key) => {
  return new Promise((resolve) => {
    resolve(crypt.decrypt(key, data));
  });
};
```

In the decryptData function, the data is in the form of an encrypted data url. This encrypted data is then decrypted using the passed in private key and returned.

## Code

The entire codebase can be found on my github [here](#). I have shown the important functions used for file upload, download, encryption, and decryption as well as those used for the generation of RSA key pairs and the AES encryption of the users RSA private key. The remainder of the code is too long to include in the form of pictures or plain text here but it is for the most part React related js files for the UI design.

Just in case the previous links to the github repository are not working the url is <https://github.com/johnmarksinclair/secure-cloud-storage> and a deployment of the application can be interacted with at <https://secure-cloud-storage.vercel.app>.