National College of Ireland

Configuration Manual

MSc Cyber Security

Vigneswaran Moorthy
Student ID: x23198311

School of Computing
National College of Ireland

Supervisor: Diego Lugones

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Vigneswaran Moorthy……

Student ID: x23198311……

Programme: MSc Cyber Security…… **Year:** …2024……..

Module: MSc Research Project……

Lecturer: Diego Lugones ……

Submission Due Date: 29/01/2025……

Project Title: Configuration Manual……

Word Count: ……1101…… **Page Count:** ……7……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Vigneswaran Moorthy……

Date: 29/01/2025……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Vigneswaran Moorthy
Student ID: x23198311

1. Introduction

In this section, the technologies and tools applied in the blockchain framework for improving security of IoT healthcare data are discussed. These include Solidity for writing smart contracts, Ganache for blockchain development; and for practical connectivity with the blockchain on Google Chrome, React.js apart from MetaMask was used for the web user interface of the application. These technologies arrive in synergy to implement role base access control, safe data sharing, and compliance with GDPR as well as HIPAA.

2. Experimental Setup

This test is performed on a personal system having the correct configurations for performing blockchain development and web interface development. Infrastructure for the system entails the hardware and software environments for Smart Contract programming that are required for the creation of a user interface for the blockchain.

- **Hardware Specifications**: Consists of an AMD Ryzen 7 @ 5700U processor and an AMD Radeon Graphics facility running at 1.80 GHz with 16 GB RAM.
- **Tools and Software**: To create a blockchain locally use Ganache, to write smart contracts use Solidity, for web app interface use React.js, and for a browser interface to the blockchain use MetaMask integrated with Google Chrome.

2. Used Technologies and Software

- **Solidity**: It is an object-based contract programming language, in which it is used for writing smart contracts which are implemented in the Ethereum Blockchain. DApps can be implemented over the blockchain where they can communicate with it and the actions performed can be made both transparent and secure using Solidity.

- **Ganache:** It is a personal blockchain utilized for Ethereum development. The local blockchain can be generated for the purpose of testing in Ganache, allowing for rapid development cycles before it's on a live network.

- **React.js:** This is a JavaScript library used in creating responsive and interactive web applications. Here, it is used for creating the user interface interacting with the blockchain and communicating with smart contracts.

- **MetaMask:** This is a browser extension which connects the user's browser to the Ethereum blockchain. It helps users securely interact with decentralized

applications (DApps) in their browsers, manage their Ethereum accounts and much more.

3. Implementation

Step 1: Setting up the Environment
Download Ganache in order to set up a development and testing blockchain environment. Install MetaMask in the browser to connect it to the blockchain with security and quickly authenticate users.

Step 2: Smart Contract Writing with Solidity
Implement the writing of smart contracts in Solidity to securely store and retrieve patients' records in a blockchain. They provide role-based access control and constitute a decentralized database for the encrypted data.

Step 3: Encrypting Sensor Data
Secure static data such as temperature, pressure, heartbeat and SpO2 data through AES encryption with safe passphrase. This step prevents data leakage and its integrity from unauthorized third parties before storing it on the blockchain.

Figure 1: Encrypt Sensor Data

Step 4: Send the data to blockchain via encryption
Share the encoded values of sensor data with the blockchain by calling the addPatientData function of the smart contract. This ensures that while communicating encrypted data is stored safely in a decentralised manner.

```
const toBlockchain = async (encrypted) => {
  let temp = CryptoES.AES.encrypt(data.feeds[0].field1, "Secret Passphrase");
  let encPlainData =
    data.feeds[0].field1.toString() +
    data.feeds[0].field2.toString() +
    data.feeds[0].field3.toString() +
    data.feeds[0].field4.toString();
  console.log("encData", encPlainData);
  let encEncryptData = CryptoES.AES.encrypt(
    encPlainData,
    "Secret Passphrase"
  ).toString();
  console.log("encEncrypt", encEncryptData);
  let encDecrypt = CryptoES.AES.decrypt(encEncryptData, "Secret Passphrase");
  console.log("encDecrypt", encDecrypt.toString(CryptoES.enc.Utf8));

  try {
    console.log("enc", encrypted);
    await sensorData.methods
      .addPatientData(
        auth._address,
        data.feeds[0].field1.toString(),
        data.feeds[0].field2.toString(),
        data.feeds[0].field3.toString(),
        data.feeds[0].field4.toString(),
        encEncryptData
      )
      .send({ from: accounts });
  } catch (e) {
    console.log(e.message);
  }
};
```

Figure 2: Send Encrypted Data to Blockchain

Step 5: Encryption Management Function: Encrypt and Transmit Information in a Blockchain

The encryptSensorData function encrypts sensor readings and lodges the encrypted data to the blockchain and its downstream systems. This function makes the encryption and submission process much simpler and more time-effective.

```
const handleEncryption = async () => {
  setEncrypted(encryptSensorData(data));
  console.log("enc", encrypted);
  toBlockchain(encrypted);
};
```

Figure 3: Encrypts the sensor data using the encryptSensorData function

Step 6: Decrypt Data from Blockchain

Back up the data as a hex string in the blockchain and then decrypt it, using the same AES passphrase, in order to get back the actual sensor data readings. This step also makes sure that only the intended people can get an opportunity of decrypting the received information.

```
const decryptSensorData = async (encrypt) => {
  console.log("decrypting values", encrypt);
  let temp = CryptoES.AES.decrypt(encrypt[0].temp, "Secret Passphrase");
  let Pressure = CryptoES.AES.decrypt(
    encrypt[1].Pressure,
    "Secret Passphrase"
  );
  let heartbeat = CryptoES.AES.decrypt(
    encrypt[2].heartbeat,
    "Secret Passphrase"
  );
  let spo2 = CryptoES.AES.decrypt(encrypt[3].spo2, "Secret Passphrase");
  let decryptedData = [
    temp.toString(CryptoES.enc.Utf8),
    Pressure.toString(CryptoES.enc.Utf8),
    heartbeat.toString(CryptoES.enc.Utf8),
    spo2.toString(CryptoES.enc.Utf8),
  ];
  console.log("decrypted", decryptedData);

  try {
    let dataFromBlock = await sensorData.methods
      .getPatientData(auth._address)
      .call();
    console.log("dcount", dataFromBlock);
  } catch (e) {
    console.log(e.message);
  }
  let decryptData = CryptoES.AES.decrypt(
    dataFromBlock[4],
    "Secret Passphrase"
  );
  console.log("dblock", decryptData);
  //return decryptedData;
};
```

Figure 4: Decrypt Data from Blockchain

Step 7: Decryption and storeSensorData Functions must be put into the Blockchain. The decryption function improves the state by populating it with readable values obtained from the retrieved blockchain data, and logs the sensor data. The process is synchronized to ensure the integrity of its contents to protect overly sensitive information.

```
const handleDecryption = () => {
  setDecrypted(decryptSensorData(encrypted));
};
const storeSensorData = (data) => {
  console.log(decrypted);
};


const dispatch = useDispatch();
```

Figure 5: Handle Decryption

Step 8: Implement State Management with Redux
Implement Redux to control the application state and perform requests, save the encrypted values and update patient's data. This helps facilitate smooth communication between the two, which are the blockchain and the frontend.

```
useEffect(() => {
  dispatch(fetchDashboard());
}, []);
useEffect(() => {
  loadWeb3();
}, []);

useEffect(() => {
  loadAccounts();
}, []);
if (status === STATUS.LOADING) {
  return <h2>Loading...</h2>;
}

if (status === STATUS.ERROR) {
  return <h2>Something Went Wrong!!!</h2>;
}
```

Figure 6: Redux State Management

Step 9: Managing Patient Sensor Data
Smart contracts such as addPatientData are responsible for storage of patients' vitals into blockchain while the health information is stored encrypted. This step makes sure that data handling is Decentralized and no one can temper with the information.

```
function addPatientData(address _patient, uint256 _temperature, uint256 _pressure, uint256 _heartBeat, uint256 _spo2 , string memory
_encryptedData) public {
    patients[_patient].temperature = _temperature;
    patients[_patient].pressure = _pressure;
    patients[_patient].heartBeat = _heartBeat;
    patients[_patient].spo2 = _spo2;
    patients[_patient].encryptedData = _encryptedData;
}
```

Figure 7: patient vitals and encrypted information on the blockchain

Step 10: Retrieve Patient Data
The getPatientData function returns the patient's records with vitals, and encrypted records for further examination. This guarantees efficient referencing of the data and at the same time enhancing its security.

```
function getPatientData(address _patient) public view returns (uint256, uint256, uint256, uint256,string memory ) {

    return (
        patients[_patient].temperature,
        patients[_patient].pressure,
        patients[_patient].heartBeat,
        patients[_patient].spo2,
        patients[_patient].encryptedData
    );
}
```

Figure 8: Retrieval of a patient's vital information and encrypted data from the blockchain