

Group Members: Archit Atrey (220195), Manvi Verma (220631)

NFC-Based HealthCare Access System

Problem Description

Healthcare access for marginalized or underserved populations is often hampered by inefficient patient identification processes and medical record management.

These populations face significant barriers including:

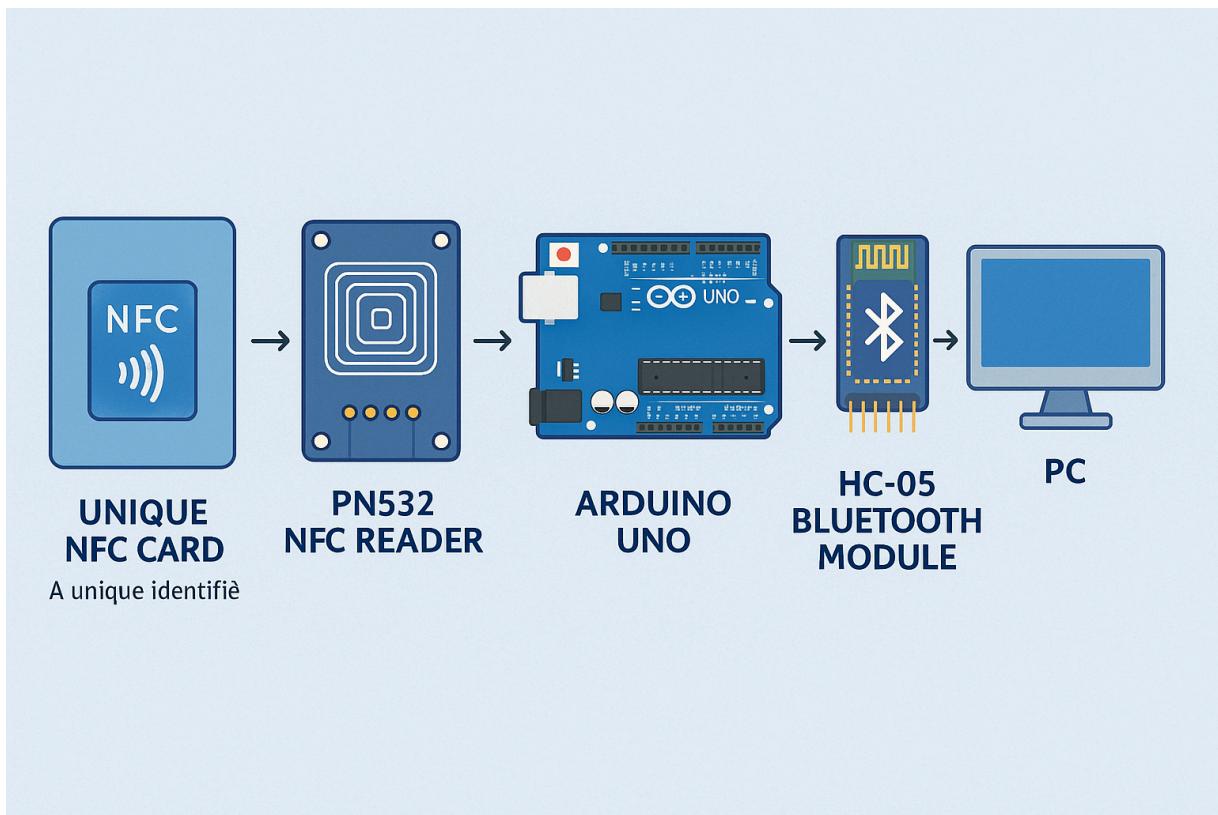
- Lack of proper identification documents, leading to difficulties in maintaining continuity of care.
- Inefficient or inaccessible medical record systems resulting in repeated tests and procedures.
- Medical errors due to unavailability of complete patient information.
- Delays in treatment caused by manual patient data retrieval.
- Limited healthcare access in remote or resource-constrained areas.

These challenges collectively lead to suboptimal healthcare delivery, misdiagnosis, delayed treatment, and reduced overall healthcare quality for vulnerable populations. The proposed system aims to address these issues through an innovative approach using widely available technology.

System Objectives

1. Create an affordable, efficient patient identification system using NFC technology
2. Enable quick and secure access to patient medical records
3. Eliminate dependence on physical documents
4. Reduce medical errors through accurate patient identification
5. Improve healthcare delivery for marginalized populations

System Block Diagram



Control Flow for Secure Patient Data Access via NFC & Bluetooth

- 1. NFC Tag Scan (User Authentication Initiation)**
 - Patient brings an **NFC tag or phone** (with UID) near the **PN532 NFC Reader**, being used in I2C mode.
 - The **Arduino** reads the **UID** from the NFC tag.
- 2. Send UID to Bluetooth Module (HC-05)**
 - Arduino sends the scanned **UID** securely to the **HC-05 Bluetooth Module**.
- 3. Transmit UID to Secure Backend System**
 - The **Bluetooth module** (HC-05) sends this UID to a **paired secure system/server** (e.g., a local secure PC or Raspberry Pi acting as the database server).
 - This system has **exclusive Bluetooth access**—not connected to the internet or any external interface for security.
- 4. Validation & Authentication**
 - The secure system receives the UID.
 - It **authenticates** the UID and checks if it is **valid and registered**.
 - If the UID is valid, it proceeds to **decrypt the corresponding encrypted**

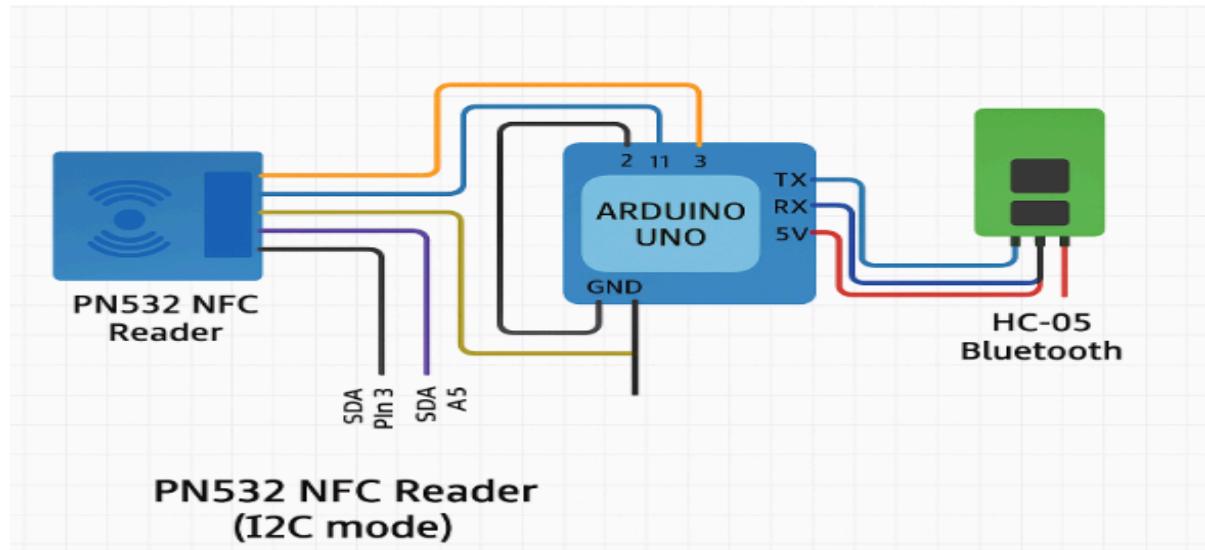
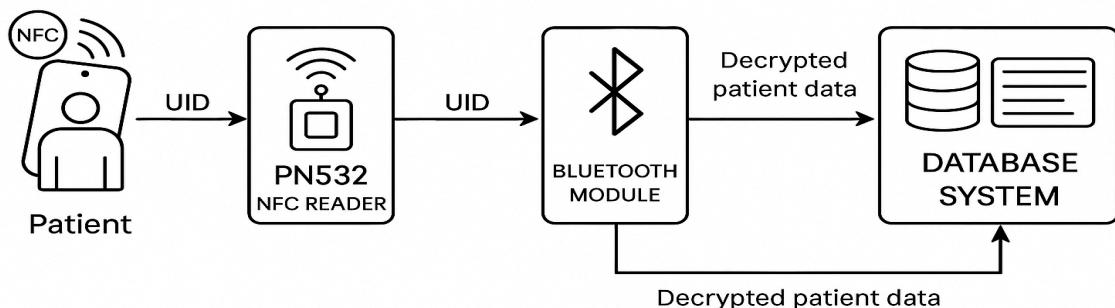
patient file using stored keys and internal logic.

5. Send Decrypted Data to Bluetooth Module

- Once decrypted, the patient's **relevant health data (file/summary)** is sent back via Bluetooth to the HC-05 module.

6. Showcase Data on Main System (Arduino UI)

- Arduino receives the decrypted patient data via serial communication from the HC-05.
- This data is then **displayed on a screen/interface** (e.g., LCD/OLED or serial monitor) connected to the Arduino.
- This ensures **only authorized access via NFC & secure Bluetooth transmission, no direct access** to keys, UIDs, or encrypted files.



Software Implementation

- 1) Check if the NFC card is detected with its UID.

Arduino Code for the same is given below:

```
arduino_nfc_bluetooth.ino
1 #include <Wire.h>
2 #include <Adafruit_PN532.h>
3 #include <SoftwareSerial.h>
4
5 // Define PN532 pins for I2C mode
6 #define PN532_IRQ 2 // Connect PN532 IRQ to Arduino digital pin 2
7 #define PN532_RESET 3 // Connect PN532 RESET to Arduino digital pin 3
8
9 // Create an instance for PN532 using I2C with IRQ and RESET
10 Adafruit_PN532 nfc(PN532_IRQ, PN532_RESET, &Wire);
11
12 // Create a SoftwareSerial instance for the HC-05 Bluetooth
13 SoftwareSerial BT(10, 11); // RX, TX for HC-05
14
15 void setup(void) {
16   Serial.begin(115200);
17   BT.begin(9600); // HC-05 default baud rate
18
19   Serial.println("Initializing PN532 (I2C mode)...");
20
21   nfc.begin();
22
23   uint32_t versiondata = nfc.getFirmwareVersion();
24   if (!versiondata) {
25     Serial.println("Didn't find PN53X board; check wiring!");
26     while (1); // halt if the PN532 is not found
27   }
28
29   Serial.print("Found chip PNS");
30   Serial.println((versiondata >> 24) & 0xFF, HEX);
31   Serial.print("Firmware ver. ");
32   Serial.print((versiondata >> 16) & 0xFF, DEC);
33   Serial.print('.');
34   Serial.println((versiondata >> 8) & 0xFF, DEC);
35
36
37
38
39
40
41 void loop(void) {
42   uint8_t uid[7]; // Buffer to store the UID
43   uint8_t uidLength; // Length of the UID
44
45   // Read an ISO14443A type tag (NFC tag or phone) using PN532
46   if (nfc.readPassiveTargetID(PN532_MIFARE_ISO14443A, uid, &uidLength)) {
47     Serial.print("Tag detected! UID: ");
48     String uidstr = "";
49     for (uint8_t i = 0; i < uidLength; i++) {
50       if (uid[i] < 0x10) Serial.print("0");
51       Serial.print(uid[i], HEX);
52       uidstr += String(uid[i], HEX);
53       Serial.print(" ");
54     }
55     Serial.println();
56
57     // Acknowledge successful scan via Bluetooth
58     BT.print("ACK: UID ");
59     BT.println(uidstr);
60
61     delay(2000); // Delay to avoid reading the same tag multiple times rapidly
62   }
63
64   // (Optional) Echo data between Serial Monitor and HC-05
65   if (BT.available()) {
66     char c = BT.read();
67     Serial.write(c);
68   }
69
70   if (Serial.available()) {
71     char c = Serial.read();
72     BT.write(c);
73   }
74 }
```

- 2) After detecting the tag, using python code we can register the new patient with its medical history details for that UID of that tag, then we can display the details of that patient using that tag.

```
view_data.py sm...
register_patient...
bin smart health...
ACK smart health...
ACK smart health...
220195.bin smart...
cli_interface.py s...
add_records.py

== Smart Healthcare CLI ==
1. Register New Patient
2. View Patient Data
3. Add Medical Records
4. Exit
Enter choice: 4

91840@ArchitAtrey MINGW64 ~/Documents/Arduino/smart healthcare system
$ python cli_interface.py

== Smart Healthcare CLI ==
1. Register New Patient
2. View Patient Data
3. Add Medical Records
4. Exit
Enter choice: 2
Waiting for NFC scan via Bluetooth...
Tag Scanned! Received UID: ACK: UID fed1c31
Access granted to UID fed1c31 for 6 hours.
Patient Data for UID fed1c31: {"name": "akarshi", "age": "17", "disease": "neck pain", "additional_records": ["more pain", "painfull"]}

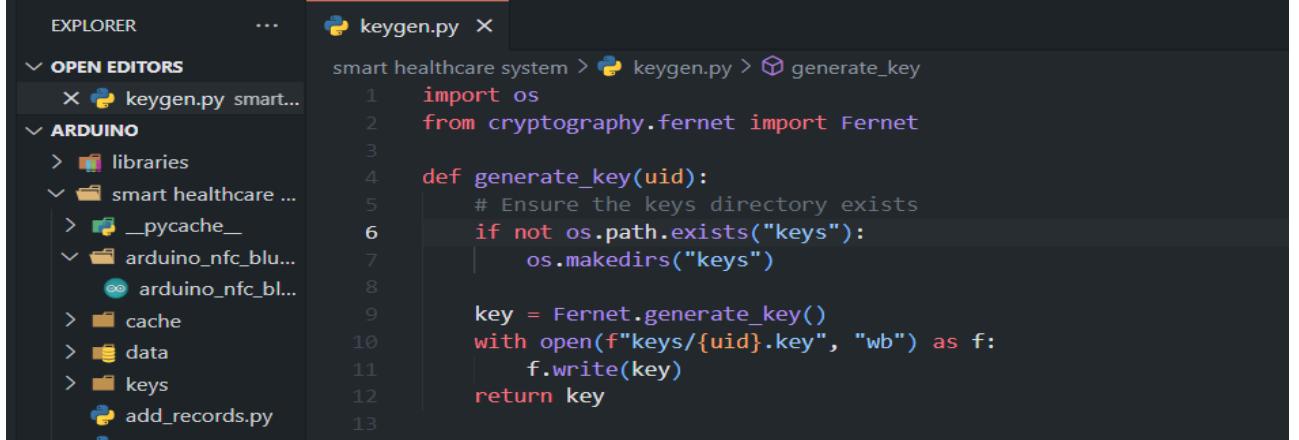
== Smart Healthcare CLI ==
1. Register New Patient
2. View Patient Data
3. Add Medical Records
4. Exit
Enter choice: 1
Waiting for NFC scan via Bluetooth...
Tag Scanned! Received UID: ACK: UID 5a60fe3
Enter patient name: vatsal
Enter patient age: 16
Enter diagnosis info: fit and fine
Error: A patient is already registered with UID 5a60fe3.

== Smart Healthcare CLI ==
1. Register New Patient
2. View Patient Data
3. Add Medical Records
4. Exit
Enter choice: 4
```

Python Backend Scripts

The following Python files live in [/Arduino/smart healthcare system/](#) and together provide patient registration, secure storage, and record-viewing functionality over Bluetooth.

keygen.py



```
keygen.py

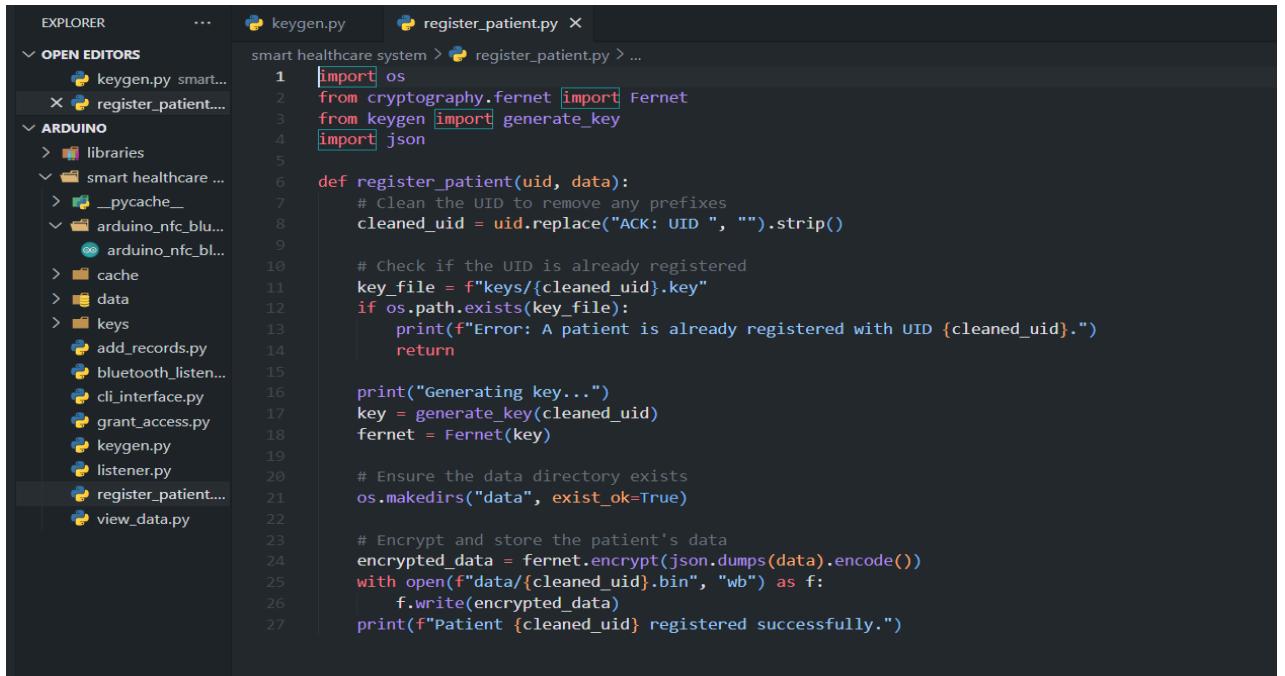
import os
from cryptography.fernet import Fernet

def generate_key(uid):
    # Ensure the keys directory exists
    if not os.path.exists("keys"):
        os.makedirs("keys")

    key = Fernet.generate_key()
    with open(f"keys/{uid}.key", "wb") as f:
        f.write(key)
    return key
```

Purpose: Generates a new symmetric encryption key for a patient's UID, saving it under [keys/{uid}.key](#). This key is used by other scripts to encrypt or decrypt that patient's data.

register_patient.py



```
register_patient.py

import os
from cryptography.fernet import Fernet
from keygen import generate_key
import json

def register_patient(uid, data):
    # Clean the UID to remove any prefixes
    cleaned_uid = uid.replace("ACK: UID ", "").strip()

    # Check if the UID is already registered
    key_file = f"keys/{cleaned_uid}.key"
    if os.path.exists(key_file):
        print(f"Error: A patient is already registered with UID {cleaned_uid}.")
        return

    print("Generating key...")
    key = generate_key(cleaned_uid)
    fernet = Fernet(key)

    # Ensure the data directory exists
    os.makedirs("data", exist_ok=True)

    # Encrypt and store the patient's data
    encrypted_data = fernet.encrypt(json.dumps(data).encode())
    with open(f"data/{cleaned_uid}.bin", "wb") as f:
        f.write(encrypted_data)
    print(f"Patient {cleaned_uid} registered successfully.")
```

Purpose: Called when a new patient is scanned and their demographic / medical-history JSON is supplied. It generates a key (via [keygen.py](#)), encrypts the JSON, and writes it to [data/{uid}.bin](#).

grant_access.py

```
import time
import json
import os

def grant_access(uid):
    # Clean the UID to remove any prefixes
    cleaned_uid = uid.replace("ACK: UID ", "").strip()

    # Check if the patient is registered by verifying the key file exists
    key_file = f"keys/{cleaned_uid}.key"
    if not os.path.exists(key_file):
        print(f"No patient registered with UID {cleaned_uid}.")
        return False # indicate failure

    # Ensure the cache directory exists
    os.makedirs("cache", exist_ok=True)

    access_data = {"uid": cleaned_uid, "timestamp": time.time()}
    with open(f"cache/{cleaned_uid}.access", "w") as f:
        json.dump(access_data, f)
    print(f"Access granted to UID {cleaned_uid} for 6 hours.")
    return True # indicate success
```

Purpose: Verifies that a scanned UID is registered, then writes a timestamped “access granted” file to `cache/`. Other scripts check `cache/` to ensure only recently-authorized requests succeed.

view_data.py

```
import json
from cryptography.fernet import Fernet
import time

def view_data(uid):
    # Clean the UID to remove any prefixes
    cleaned_uid = uid.replace("ACK: UID ", "").strip()

    # Check if the key file exists
    key_file = f"keys/{cleaned_uid}.key"
    if not os.path.exists(key_file):
        print(f"No patient registered with UID {cleaned_uid}.")
        return

    # Check if access is granted
    access_file = f"cache/{cleaned_uid}.access"
    if not os.path.exists(access_file):
        print("Access not granted.")
        return

    # Verify access time
    with open(access_file, "r") as f:
        access_data = json.load(f)
        if time.time() - access_data["timestamp"] > 6 * 3600:
            print("Access expired.")
            return

    # Decrypt and display patient data
    try:
        with open(key_file, "rb") as f:
            key = f.read()
            cipher = Fernet(key)

        with open(f"data/{cleaned_uid}.bin", "rb") as f:
            encrypted_data = f.read()

        decrypted_data = cipher.decrypt(encrypted_data).decode()
        print(f"Patient Data for UID {cleaned_uid}: {decrypted_data}")
    except FileNotFoundError as e:
        print(f"Error: {e}")
```

Purpose: After access has been granted, this script decrypts the patient’s stored JSON data and prints it to the console.

bluetooth_listener.py

The screenshot shows the VS Code interface with the 'bluetooth_listener.py' file open in the editor tab. The code implements a function to wait for a Bluetooth connection and another to listen for NFC tags via serial communication.

```
1 import serial
2 import time
3
4 def wait_for_uid():
5     # Wait until the Bluetooth connection is available.
6     while True:
7         try:
8             print("Waiting for Bluetooth connection to HC05 on COM10...")
9             ser = serial.Serial('COM10', 9600, timeout=10)
10            break # If the port opened successfully, exit the loop.
11        except serial.serialutil.SerialException as e:
12            print("Bluetooth not connected. Retrying in 5 seconds...")
13            time.sleep(5)
14
15    # Now wait for an NFC tag to be scanned.
16    while True:
17        line = ser.readline().decode().strip()
18        if line:
19            print(f"✓ Tag Scanned! Received UID: {line}")
20            return line
21        else:
22            print("⚠ No UID received. Make sure the NFC tag is close enough.")
```

Purpose: Blocks until the HC-05 Bluetooth link is up, then reads and returns the next NFC-UID string sent by the Arduino.

listener.py

The screenshot shows the VS Code interface with the 'listener.py' file open in the editor tab. This script defines a function to listen on a serial port for NFC UIDs and immediately call a 'grant_access' function to authorize them.

```
1 import serial
2 from grant_access import grant_access
3
4 def listen_bluetooth(port):
5     ser = serial.Serial(port, 9600)
6     print("Listening for UID...")
7
8     while True:
9         if ser.in_waiting:
10            uid = ser.readline().decode().strip()
11            print(f"Received UID: {uid}")
12            grant_access(uid)
13
```

Purpose: A simple daemon that listens on a serial port for incoming UIDs and immediately calls `grant_access(uid)` to authorize them.

add_records.py

```
EXPLORER ... listener.py add_records.py
smart healthcare system > add_records.py > ...
OPEN EDITORS
  listener.py smart...
  add_records.py...
ARDUL... D E U
  libraries
  smart healthcare ...
    _pycache_
    arduino_nfc_blu...
      arduino_nfc_blu...
    cache
    data
    keys
    add_records.py
    bluetooth_listen...
    cli_interface.py
    grant_access.py
    keygen.py
    listener.py
    register_patient...
    view_data.py
1 import os
2 import json
3 from cryptography.fernet import Fernet
4
5 def add_medical_records(uid):
6     # Clean the UID to remove any prefixes
7     cleaned_uid = uid.replace("ACK: UID ", "").strip()
8
9     # Check if the key file exists
10    key_file = f"keys/{cleaned_uid}.key"
11    if not os.path.exists(key_file):
12        print(f"No patient registered with UID {cleaned_uid}.")
13        return
14
15    # Check if the data file exists
16    data_file = f"data/{cleaned_uid}.bin"
17    if not os.path.exists(data_file):
18        print(f"No medical records found for UID {cleaned_uid}.")
19        return
20
21    # Decrypt the existing data
22    try:
23        with open(key_file, "rb") as f:
24            key = f.read()
25            cipher = Fernet(key)
26
27        with open(data_file, "rb") as f:
28            encrypted_data = f.read()
29
30            decrypted_data = json.loads(cipher.decrypt(encrypted_data).decode())
31    except Exception as e:
32        print(f"Error decrypting data: {e}")
33        return
```

```
EXPLORER ... listener.py add_records.py
smart healthcare system > add_records.py > ...
OPEN EDITORS
  listener.py smart...
  add_records.py...
ARDUL... D E U
  libraries
  smart healthcare ...
    _pycache_
    arduino_nfc_blu...
      arduino_nfc_blu...
    cache
    data
    keys
    add_records.py
    bluetooth_listen...
    cli_interface.py
    grant_access.py
    keygen.py
    listener.py
    register_patient...
    view_data.py
5 def add_medical_records(uid):
34
35    # Display existing data
36    print(f"Existing Medical Records for UID {cleaned_uid}: {decrypted_data}")
37
38    # Add new medical records
39    new_disease = input("Enter new diagnosis info: ")
40    if "additional_records" not in decrypted_data:
41        decrypted_data["additional_records"] = []
42        decrypted_data["additional_records"].append(new_disease)
43
44    # Re-encrypt and save the updated data
45    try:
46        updated_data = cipher.encrypt(json.dumps(decrypted_data).encode())
47        with open(data_file, "wb") as f:
48            f.write(updated_data)
49        print(f"New medical records added for UID {cleaned_uid}.")
50    except Exception as e:
51        print(f"Error saving updated data: {e}")
```

Purpose: Loads and decrypts existing patient data, prompts the user to enter new medical-history items, appends them, then re-encrypts and saves back to **data/{uid}.bin**.

cli_interface.py

```
EXPLORER ... cli_interface.py X smart healthcare system > cli_interface.py > ...
1 import os
2 from register_patient import register_patient
3 from view_data import view_data
4 from grant_access import grant_access
5 from bluetooth_listener import wait_for_uid
6 from add_records import add_medical_records
7
8 def menu():
9     while True:
10         print("\n==== Smart Healthcare CLI ===")
11         print("1. Register New Patient")
12         print("2. View Patient Data")
13         print("3. Add Medical Records")
14         print("4. Exit")
15
16         choice = input("Enter choice: ")
17
18         if choice == "1":
19             uid = wait_for_uid()
20             cleaned_uid = uid.replace("ACK: UID ", "").strip()
21             # Check if the patient is already registered by checking for the key file
22             if os.path.exists(f"keys/{cleaned_uid}.key"):
23                 print(f"Patient with UID {cleaned_uid} is already registered.")
24             else:
25                 name = input("Enter patient name: ")
26                 age = input("Enter patient age: ")
27                 disease = input("Enter diagnosis info: ")
28                 data = {"name": name, "age": age, "disease": disease}
29                 register_patient(uid, data)
30
31         elif choice == "2":
32             uid = wait_for_uid() # Get UID via Bluetooth
33             if grant_access(uid): # Only view data if access is granted
34                 view_data(uid) # View patient data
35
36         elif choice == "3":
37             uid = wait_for_uid() # Get UID via Bluetooth
38             if grant_access(uid): # Grant access for the scanned UID
39                 add_medical_records(uid) # Add additional medical records
40
41         elif choice == "4":
42             break
43         else:
44             print("Invalid choice.")
45
46 if __name__ == "__main__":
47     menu()
```

Purpose: Provides a single-entry command-line menu that ties together all functionality: scanning a UID over Bluetooth, then registering, viewing, or updating a patient's encrypted record.

How It All Fits Together:

1. **Arduino → HC-05** sends raw UID over Bluetooth when an NFC tag/phone is tapped.
2. **bluetooth_listener.py** picks up that UID and returns it.
3. **cli_interface.py** drives the workflow: on “Register” it calls **register_patient.py**; on “View”/“Add” it first calls **grant_access.py** then either **view_data.py** or **add_records.py**.
4. **keygen.py** underpins secure storage by generating per-patient keys.

5. All JSON patient data lives encrypted in `data/`; encryption keys in `keys/`; temporary access flags in `cache/`.

Encryption Mechanism & File-Storage Formats

Encryption Scheme

All patient files are encrypted using the **Fernet** implementation from the Python `cryptography` library. Fernet provides:

- **Symmetric AES-128 in CBC mode** under the hood
- **Per-message IV**: a fresh 16-byte random IV on every encryption
- **HMAC-SHA256** for integrity/authenticity over the entire payload
- **Timestamping**: embeds an 8-byte UNIX timestamp so you can later enforce expiration if desired
- **Single URL-safe Base64 token** that concatenates:
 1. Version byte
 2. Timestamp
 3. IV
 4. Ciphertext
 5. HMAC signature

Because the final token is base64-encoded, it's easy to store as text or binary on disk.

Key Files

- Location:

```
keys/{UID}.key
```

Content: the 32-byte Fernet key, URL-safe Base64-encoded (44 ASCII chars + newline).

Handling: created by `keygen.py`, tested for existence by every reader/writer.

Encrypted Data Files

- Location:

```
data/{UID}.bin
```

Content: the raw bytes of `Fernet.encrypt(...)`—i.e. a URL-safe Base64 string, ~300–400 bytes for a small JSON.

Format: binary mode write, but file contents are ASCII Base64. Decrypted back to the original JSON with `Fernet.decrypt()`.

Cache Files (Access Tokens)

- Location:

```
cache/{UID}.access
```

Content: plain-text JSON, e.g.

```
{"uid": "5a60fe3", "timestamp": 1711109732.12345}
```

Purpose: track when `grant_access()` was last invoked so you can enforce a time-window (e.g. 6 hours) on subsequent reads or record additions.

Raw Patient Data (Pre-Encryption)

Always a JSON-serializable Python `dict`, for example:

```
{  
    "name": "Archit Atrey",  
    "age": 23,  
    "history": ["hypertension", "allergy: penicillin"],  
    "additional_records": []  
}
```

This is dumped via `json.dumps(...)` before being encrypted, and re-loaded after decrypting.

System Architecture

The proposed healthcare access system consists of four main components:

1. Patient Identification Module:
 - Uses NFC-enabled smartphones instead of traditional RFID tags
 - Stores a unique patient identifier in the phone's NFC chip
 - Eliminates need for physical cards or wristbands
2. Data Reading Module:
 - PN532 NFC Reader connected to Arduino UNO
 - Reads patient identification data from NFC-enabled phones
 - Processes the information for transmission
3. Data Transmission Module:
 - HC-05 Bluetooth module for wireless communication
 - Transmits patient identification data to healthcare provider's device
 - Enables local operation without requiring internet connectivity
4. Database Management Module:
 - Stores comprehensive patient health records
 - Links records to unique patient identifiers
 - Provides secure access to authorized healthcare providers

Key Results

The implementation of this system is expected to deliver the following key outcomes:

1. Improved Patient Identification: Elimination of patient misidentification errors through unique NFC identifiers.
2. Enhanced Efficiency: Reduction in patient registration and record retrieval time from minutes to seconds.
3. Reduced Barriers to Care: Removal of dependency on physical identification documents, particularly benefiting marginalized populations.
4. Better Continuity of Care: Complete medical history available to healthcare providers at point of care.
5. Cost-Effective Solution: Utilization of widely available technology (smartphones, Arduino) reduces implementation costs compared to proprietary systems.

Conclusion

The RFID/NFC-Based Healthcare Access System offers a practical, affordable solution to address healthcare access challenges faced by marginalized populations. By utilizing widely available technology (smartphones, Arduino components), the system overcomes barriers to healthcare while ensuring security and privacy of patient information.

The proof-of-concept demonstrates that NFC-enabled mobile phones can effectively replace dedicated RFID tags, making the system more accessible and user-friendly. Bluetooth-based communication enables operation in areas with limited connectivity, ensuring that healthcare facilities in underserved areas can benefit from this technology.

This project directly addresses the needs of marginalized populations by simplifying patient identification, streamlining healthcare delivery, and reducing dependence on physical documentation. The system's implementation could significantly improve healthcare outcomes for vulnerable groups by ensuring continuous, error-free access to medical care.