

4-2020

## **Sdhcare: Secured Distributed Healthcare System**

Mohammed R. S. Al Baqari

Follow this and additional works at: [https://scholarworks.uaeu.ac.ae/info\\_sec\\_theses](https://scholarworks.uaeu.ac.ae/info_sec_theses)



Part of the [Information Security Commons](#)

---

### **Recommended Citation**

Al Baqari, Mohammed R. S., "Sdhcare: Secured Distributed Healthcare System" (2020). *Information Security Theses*. 6.

[https://scholarworks.uaeu.ac.ae/info\\_sec\\_theses/6](https://scholarworks.uaeu.ac.ae/info_sec_theses/6)

This Thesis is brought to you for free and open access by the Information Security at Scholarworks@UAEU. It has been accepted for inclusion in Information Security Theses by an authorized administrator of Scholarworks@UAEU. For more information, please contact [fadl.musa@uaeu.ac.ae](mailto:fadl.musa@uaeu.ac.ae).

United Arab Emirates University

College of Information Technology

Department of Information Systems and Security

**SDHCARE: SECURED DISTRIBUTED HEALTHCARE SYSTEM**

Mohammed R. S. Al Baqari

This thesis is submitted in partial fulfilment of the requirements for the degree of  
Master of Science in Information Security

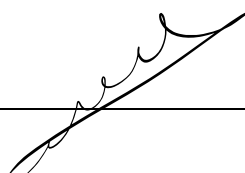
Under the Supervision of Dr. Ezedin Baraka

April 2020

### Declaration of Original Work

I, Mohammed R. S. Al Baqari, the undersigned, a graduate student at the United Arab Emirates University (UAEU), and the author of this thesis entitled “*SDHCARE: Secure Distributed Healthcare System*”, hereby, solemnly declare that this thesis is my own original research work that has been done and prepared by me under the supervision of Dr. Ezedin Baraka, in the College of Information Technology (CIT) at UAEU. This work has not previously been presented or published, or formed the basis for the award of any academic degree, diploma or a similar title at this or any other university. Any materials borrowed from other sources (whether published or unpublished) and relied upon or included in my thesis have been properly cited and acknowledged in accordance with appropriate academic conventions. I further declare that there is no potential conflict of interest with respect to the research, data collection, authorship, presentation and/or publication of this thesis.

Student's Signature: \_\_\_\_\_



Date: 02/07/2020

Copyright © 2020 Mohammed R. S. Al Baqari  
All Rights Reserved



## Approval of the Master Thesis


This Master Thesis is approved by the following Examining Committee Members:

- 1) Advisor (Committee Chair): Ezedin Barka

Title: Associate Professor

Department of Information Systems and Security

College of Information Technology

Signature 

Date 23/4/2020

- 2) Member: Khaled Shuaib

Title: Professor

Department of Information Systems and Security

College of Information Technology

Signature 

Date 23/4/2020

- 3) Member (External Examiner): Khaled Salah

Title: Professor

Department of Electrical Engineering and Computer Science

Institution: Khalifa University- Abu Dhabi, UAE

Signature 

Date 23/4/2020

This Master Thesis is accepted by:

Dean of the College of Information Technology: Professor Taieb Znati

Signature \_\_\_\_\_ Date 02/07/2020

Dean of the College of Graduate Studies: Professor Ali Al-Marzouqi

Signature \_\_\_\_\_ Date 05/07/2020

Copy \_\_\_\_ of \_\_\_\_

## Abstract

In healthcare sector, the move towards Electronic Health Records (EHR) systems has been accelerating in parallel with the increased adoption of IoT and smart devices. This is driven by the anticipated advantages for patients and healthcare providers. The integration of EHR and IoT makes it highly heterogeneous in terms of devices, network standard, platforms, types data, connectivity, etc. Additionally, it introduces security, patient and data privacy, and trust challenges. To address such challenges, this thesis proposes an architecture that combines biometric-based blockchain technology with the EHR system. More specifically, this thesis describes a mechanism that uses patient's fingerprint for recovery of patient's access control on their EHRs securely without compromising their privacy and identity. A secure distributed healthcare system (SDHCARE) is proposed to uniquely identify patients, enable them to control access to, and ensure recoverable access to their EHRs that are exchanged and synchronized between distributed healthcare providers. The system takes into account the security and privacy requirements of Health Insurance Portability and Accountability Act (HIPAA) compliance, and it overcomes the challenges of using secret keys as patient's identity to control access to EHRs. The system used distributed architecture with two layers being local to each healthcare provider that is member of SDHCARE, and two layers shared across all members of SDHCARE system. SDHCARE system was prototyped and implemented in order to validate its functional requirements, security requirements and to evaluate its performance. The results indicated successful fulfillment of design requirements without significant overhead on the performance as required by healthcare environments.

**Keywords:** Blockchain, Healthcare, EHR, Fingerprint, Biometric, Access Control.

## Title and Abstract (in Arabic)

### نظام الرعاية الصحية الموزع الآمن

#### الملخص

في قطاع الرعاية الصحية، كان التحرك نحو أنظمة السجلات الصحية الإلكترونية (EHR) يتسارع بالتوازي مع زيادة اعتماد إنترنت الأشياء والأجهزة الذكية، وهذا مدفوع بالمزايا المتوقعة للمرضى ومقدمي الرعاية الصحية، وعلى الرغم من المزايا المرجوة فإن دمج السجلات الصحية الإلكترونية مع إنترنت الأشياء يجعلها غير متجانسة للغاية من حيث الأجهزة، ومعايير الشبكة، والأنظمة الأساسية، وأنواع البيانات، والاتصال، وما إلى ذلك، إضافة إلى خصوصية المريض والبيانات، وتحديات الثقة، ولمواجهة هذه التحديات، تقترح هذه الأطروحة بنية تجمع بين تقنية البلوكتشين المستندة إلى المقاييس الحيوية ونظام EHR. تحديدًا، تصف هذه الأطروحة آلية تستخدم بصمة المريض لاستعادة التحكم في سجلاته الصحية الخاصة EHR بأمان ودون المساس بخصوصيته وهويته، حيث يقترح نظام رعاية صحية موزع آمن (SDHCARE) لتحديد المرضى بشكل فريد، وتمكينهم من التحكم في الوصول إلى سجلاتهم الصحية، وضمان الوصول القابل للاسترداد إلى السجلات الصحية الإلكترونية الخاصة بهم التي يتم تبادلها ومزامنتها بين مقدمي الرعاية الصحية الموزعة، و يأخذ النظام في الاعتبار متطلبات الأمان والخصوصية للامتثال لقانون قابلية التأمين الصحي والمساءلة (HIPAA)، ويتغلب على تحديات استخدام المفاتيح السرية كهوية المريض للتحكم في الوصول إلى السجلات الصحية الإلكترونية. ويستخدم النظام بنية موزعة مع طبقتين محليتين لكل مقدم رعاية صحية عضو في SDHCARE، وطبقتين مشتركتين عبر جميع أعضاء نظام SDCHARE، حيث تم تصميم النظام وتطبيقه من أجل التحقق من متطلباته الوظيفية ومتطلبات الأمان وتقييم أدائه، وأشارت النتائج إلى تحقيق متطلبات التصميم بنجاح دون أعباء كبيرة على الأداء كما هو مطلوب في بيئات الرعاية الصحية.

**مفاهيم البحث الرئيسية:** البلوكتشين، والرعاية الصحية، EHR، بصمات الأصابع، البيومترية، التحكم في الوصول.

## **Acknowledgements**

I would like to express my sincere gratitude to my advisor Dr. Ezedin Barak for his continuous support of my master's study and research and for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me as I researched and wrote this thesis. I could not have imagined a better advisor and mentor for my master's study.

Besides my advisor, I would like to thank the rest of my thesis committee, Prof. Khaled Shuaib and Dr. Khalid Saleh, for their encouragement and insightful feedback. In addition, I would like to thank all of the members of the Information Security Department of the College of IT of United Arab Emirates University for assisting me with my studies and research. My special thanks are extended to the Library Research Desk for providing me with relevant reference materials.

Special thanks go to my parents, brothers, and sisters who helped me along the way. I am sure they suspected it was endless.

## **Dedication**

*To my beloved parents and family*

## Table of Contents

Title .....	i
Declaration of Original Work .....	ii
Copyright .....	iii
Approval of the Master Thesis .....	iv
Abstract .....	vi
Title and Abstract (in Arabic) .....	vii
Acknowledgements .....	viii
Dedication .....	ix
Table of Contents .....	x
List of Tables .....	xii
List of Figures .....	xiii
List of Abbreviations .....	xv
Chapter 1: Introduction .....	1
1.1 Overview .....	1
1.2 Statement of the Problem .....	5
1.3 Research Objectives .....	6
1.4 Research Methodology .....	6
1.5 Literature Review .....	9
1.5.1 Blockchain Basics .....	10
1.5.2 Blockchain Security .....	16
1.5.3 Blockchain in Healthcare .....	24
Chapter 2: Methods .....	38
2.1 Design Overview .....	38
2.1.1 User Interface (UI) Layer .....	39
2.1.2 Middleware Layer .....	41
2.1.3 Blockchain Layer .....	43
2.1.4 Cloud Store Layer .....	48
2.2 Functional Use Cases .....	48
2.2.1 SDHCARE Provider Enrollment .....	48
2.2.2 Patient Registration .....	50
2.2.3 Patient Appointment Management .....	52
2.2.4 Doctor's EHR Access .....	55
Chapter 3: Implementation .....	61
3.1 Prototype Components .....	61
3.2 Django Architecture .....	63

3.3 System Implementation .....	65
3.3.1 Building the Runtime Environment .....	65
3.3.2 Initializing SDHCARE Web and Database Components .....	67
3.3.3 Building SDHCARE Django Code .....	70
3.3.4 Building Ethereum Smart Contracts .....	83
3.3.5 Integrating Django with Ethereum .....	87
Chapter 4: Testing and Performance Evaluation .....	91
4.1 Functional Testing .....	91
4.2 Security Testing .....	93
4.3 Performance Evaluation .....	99
Chapter 5: Conclusion .....	104
References .....	106
Appendix .....	111



## List of Tables

Table 1: Summary of EHR Requirements and Blockchain Opportunities .....	3
Table 2: Summary of the Literature .....	10
Table 3: Factors for Blockchain Transaction Delay Variation .....	21
Table 4: Access Control Matrix by accessControlSC.....	47
Table 5: Python Libraries Required for SDHCARE.....	67
Table 6: Summary of Views Functions .....	77
Table 7: Summary of Forms Functions .....	79
Table 8: Summary of Tables Functions .....	79

## List of Figures

Figure 1: Percentage of EHR Systems Adoption in US .....	2
Figure 2: General Chain of Block.....	11
Figure 3: Block Structure.....	13
Figure 4: The Effects of Tampering with One Block in the Ledger .....	19
Figure 5: Eclipse Attack.....	24
Figure 6: SDHCARE High-Level Architecture .....	39
Figure 7: Summary of providersTable Data .....	44
Figure 8: Summary of patientsTable Data .....	45
Figure 9: Summary of ehrHashTable Data .....	46
Figure 10: Hospital Enrollment Sequence Diagram .....	50
Figure 11: Summary of the Patient Registration Process .....	52
Figure 12: Process for Booking a New Appointment.....	53
Figure 13: Booking Confirmation and Access Granting Summary.....	55
Figure 14: Summary of the EHR Read Process.....	58
Figure 15: Summary of the EHR Write Process .....	60
Figure 16: Django Web Development Architecture .....	63
Figure 17: MacOS Software Version for SDHCARE .....	65
Figure 18: PyCharm SDHCARE Project with Python 3.8 Interpreter.....	66
Figure 19: Example of Users, Groups, and Group Assignment .....	68
Figure 20: Example of SDHCARE Databases.....	70
Figure 21: Summary of HTML SDHCARE Template Structure .....	71
Figure 22: Admin UI with Forms and Views .....	72
Figure 23: Doctor UI with Forms and View.....	72
Figure 24: Reception UI with Forms and Views .....	73
Figure 25: Healthcare Provider's getProviderInfo Page.....	74
Figure 26: SDHCARE Home Page.....	80
Figure 27: SDHCARE About Page.....	80
Figure 28: SDHCARE Login Page.....	81
Figure 29: SDHCARE Admin UI Page .....	81
Figure 30: SDHCARE Admin UI Submit Page.....	82
Figure 31: SDHCARE-Reception UI Page.....	82
Figure 32: SDHCARE Doctor UI Page .....	83
Figure 33: Summary of Ethereum Transactions .....	86
Figure 34: Integrating Django with Ethereum .....	87
Figure 35: Infura Account Details .....	89
Figure 36: Custom Python Module Operation.....	90
Figure 37: Synchronization of Patient's EHR .....	92
Figure 38: Verifying EHR Recovery Using Patient ID .....	92
Figure 39: Verifying Hash as Names in Azure Files .....	93
Figure 40: Failed Login to Reception Portal using a Doctor Account .....	94

Figure 41: Fingerprint Validation before Writing an EHR.....	95
Figure 42: Failed Attempt to Write a New EHR .....	95
Figure 43: Failed Attempt to Read a Patient's HER.....	96
Figure 44: Sample Audit Logs for New EHR.....	97
Figure 45: Sample of Anonymized Data Store in Azure Files .....	98
Figure 46: Performance Measurement using Google Chrome .....	100
Figure 47: Read Performance Testing .....	101
Figure 48: Write Performance Testing .....	101
Figure 49: Read Time Analysis with Larger ehrHashTable .....	103

## List of Abbreviations

ABI	Application Binary Interface
API	Application Programmable Interface
BBDS	Blockchain-Based Data Sharing
CA	Certificate Authority
CIA	Confidentiality, Integrity, and Availability
DOA	Decentralized Autonomous Organization
DOB	Date of Birth
DOS	Denial of Service
EHR	Electronic Health Record
FHIR	Fast Healthcare Interoperability Resources
GETH	Go Ethereum
HIPAA Act	Health Insurance Portability and Accountability Act [US]
HITECH Act	Health Information Technology for Economic and Clinical Health Act [US]
HL7	Health Level Seven
IoT	Internet of Things
IP	Internet Protocol
IT	Information Technology
NHA	National Health Authority
NHS	National Health Service [UK]
NID	National Identity
NIST	National Institute of Standardization and Technology
OHP	Owner Healthcare Provider

ONC	Office of the National Coordinator
OS	Operating System
PACL	Patient Access Control List
PII	Personal Identification Information
POA	Proof of Authority
POW	Proof of Work
POS	Proof of Stack
POV	Proof of Verification
PPG	Photo Plethysmo Gram
RBAC	Role-Based Access Control
UAEU	United Arab Emirates University
UI	User Interface
U.S.	United States of America

## Chapter 1: Introduction

### 1.1 Overview

In the healthcare sector, the move towards electronic health record (EHR) systems has been accelerating in parallel with the increased adoption of IoT and smart devices. This is driven by the anticipated advantages for patients and healthcare providers. The Office of the National Coordinator (ONC) for Health Information Technology within the U.S. Department of Health and Human Services [1], defined EHR as “a digital version of a patient’s paper chart. EHRs are real-time, patient-centered records that make information available instantly and securely to authorized users. While an EHR does contain the medical and treatment histories of patients, an EHR system is built to go beyond standard clinical data collected in a provider’s office and can be inclusive of a broader view of a patient’s care. They are built to share information with other health care providers, such as laboratories and specialists, so they contain information from all the clinicians involved in the patient’s care”.

Some advantages of EHR systems as indicated by ONC [1] are:

- They maintain and synchronize patients’ medical history, diagnoses, medications, treatment plans, immunization dates, allergies, radiology images, and laboratory and test results
- They allow access to evidence-based tools that providers can use to make decisions about patients’ care
- They automate and streamline provider workflow

In 2009, the U.S. government passed the Health Information Technology for Economic and Clinical Health Act (HITECH Act) to motivate the adoption and

meaningful use of EHR systems. Additionally, they included this objective in the Federal Health IT Strategic Plan [2]. Figure 1 represents the progress of EHR system adoption in the United States since 2001.

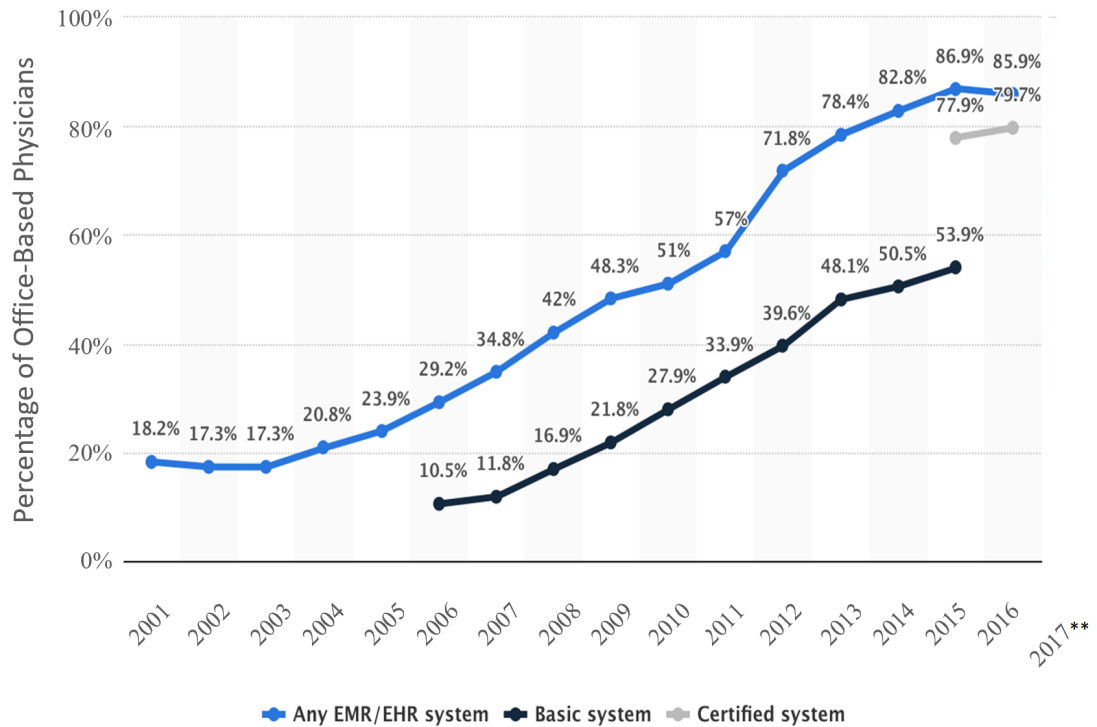


Figure 1: Percentage of EHR Systems Adoption in US

In parallel with the move towards EHR healthcare systems, Blockchain technology was introduced by Satoshi Nakamoto in 2009 [3]. Blockchain technology has received significant attention from the research community. Blockchain's decentralized nature along with its cryptographic services increased its potential to be the future platform for distributed systems. The initial phase of Blockchain technology was limited to the financial sector and mainly focused on cryptocurrency such as Bitcoin, which has evolved to become the most popular cryptocurrency application.

Over time, Blockchain technology has evolved with the introduction of Ethereum and its smart contracts capability Buterin [4], which provides the programmability component of Ethereum Blockchain. This step gave Blockchain technology huge potential and expanded its scope from the financial to other sectors, including healthcare, education, government, and manufacturing. In particular, for the healthcare sector and EHR systems, Blockchain technology offers many capabilities that can fulfill several EHR requirements, as described by McGhin, Choo, Liu, and He [5], and summarized in Table 1.

Table 1: Summary of EHR Requirements and Blockchain Opportunities

<b>EHR Requirements</b>	<b>Blockchain Opportunities</b>
System security, including authentication, integrity, access control, and non-repudiation for multiparty integrated EHR systems	Security, assurance, and immutability are provided using cryptography, namely private and public keys combined with hash-chaining between blocks of data
Interoperability between different EHR standards implemented by various healthcare providers, research entities, insurance providers, and pharmacies	Smart contracts capability provides an abstraction layer to enable communication among miners in distributed healthcare providers running different EHR standards
Data sharing of health records	The decentralized architecture allows multiple entities to share health records
Mobility of healthcare systems with the introduction of IoT and smart devices that allow patients to share and access their health records	Shared data across distributed ledgers enables near real-time updates across the network to all parties
Availability of the healthcare system	The technology provides high availability and resilience through its decentralized model of operation



The close convergence between EHR requirements and Blockchain capabilities listed in Table 1 is a key driver that has led many proposals to include Blockchain-based EHR applications, including EHR monitoring and auditing, mobility applications, and exchange of information.

Despite the promising convergence between Blockchain and EHR systems, some limitations have been identified regarding the integration between Blockchain and EHR systems, specifically:

- Key management in current implementations of Blockchain are based on private/public key pairs. In Blockchain-based EHR systems, patients that use their private keys as their identities to control access to and to sign their EHRs are subject to permanent loss of access to their records in the case of lost private keys, as discussed in McGhin [5]. This is because, in asymmetric cryptography, a private key is not recoverable from the public key (computationally infeasible).
- The lack of standardization for various deployments of Blockchain in healthcare systems generates a challenge regarding the interoperability and exchange of EHRs, which limits the success of deployments [5].
- There is the potential for privacy leakage due to the unencrypted nature of blocks that hold information related to patients' health. Even with encrypted blocks, the ability to access the blocks publicly, in public ledgers, makes them subject to cryptanalysis attacks, which can exploit patients' privacy if the encryption algorithms are compromised [5].
- Scalability and IoT overhead can occur due to the increased number of medical IoT devices and medical sensors joining the Blockchain

network. The more IoT devices joining the Blockchain network, the more the computational complexity of the ledger, which leads to the need for more computational power on these IoT devices. However, these IoT devices have very limited computational capabilities and are not designed to support the complex operations required by Blockchain hashing algorithms [5].

- Blockchain entails security vulnerabilities, which are discussed in detail in Section 1.5.2.

Accordingly, the scope of the present thesis focuses on investigating solutions to overcome the privacy leakage and key recovery challenges. Specifically, the focus is on finding a solution that enables blockchain-based synchronization and exchange of EHRs between distributed healthcare providers while:

- Ensuring recoverable access to patients' EHRs
- Maintaining the unique patients' identities in EHRs
- Controlling access to EHRs
- Providing anonymity to patients' EHRs in the datastores
- Detecting any modifications to EHRs, and
- Logging any activities on EHRs.

## **1.2 Statement of the Problem**

There are several limitations when deploying Blockchain technology in the healthcare industry, as detailed in Section 1.1. The problem statement targeted by this research is patients' inability to recover access to their EHR records in the case of missing patient identities. This can occur in many cases such as lost private keys or emergency access to EHRs by unauthorized healthcare providers.

### **1.3 Research Objectives**

The core objective of this research project is to propose an EHR exchange and synchronization system based on Blockchain that can provide patients with an access recovery mechanism to their EHRs. In addition to this objective, the proposed system fulfills the requirements of Health Insurance Portability and Accountability Act (HIPAA Act). These legislative requirements include the authentication of access to EHRs, maintenance of EHR isolation and privacy for each patient, guaranteed integrity of EHRs, governance of access control to EHRs, and guaranteed audit logs for patient's EHRs.

### **1.4 Research Methodology**

This section describes the methodology followed to conduct the research project. The methodology was structured in a logical order to enable gaining in-depth knowledge about the research topic, and divided into several phases:

- A literature review to understand the status of current research on Blockchain-based EHR systems and identify areas for research.
- Verification of the uniqueness of the proposal to address the issues presented in the statement of the problem.
- Development of high-level design architecture of the proposed solution followed by a low-level design.
- A prototype of the proposed design, using Python and Ethereum, to simulate the functional and security requirements.
- Analysis of the strengths and weaknesses of the proposed solution from functional and security perspectives.

Google Scholar was used to search several bibliographic databases for literature related to the current state of research on blockchain in healthcare. The search string deployed was:

Blockchain AND (Healthcare OR EHR) AND (source: ieee OR source: springer OR source: acm OR source: sage OR source: Elsevier).

The output from Google Scholar identified 3,390 papers, of which 3,210 papers were published since 2015.

Initially, only review papers were considered to obtain an overall understanding of current research. Only top five cited papers were considered for the literature review. These papers were obtained using the UAEU E-Library, which provides access to the selected databases used in the Google Scholar filter.

From the initial results, the identified papers were classified into three main categories:

- Blockchain core technology – Top five cited papers
- Blockchain security – Top five cited papers
- Blockchain in healthcare – Top 10 cited papers

From this review, limitations in the existing blockchain-based EHR proposals were identified, forming the scope of research for the present thesis.

Google Scholar was consulted again to identify literature related to the use of biometrics in Blockchain deployment in healthcare. One paper was identified that details the use of Fuzzy Vault combined with photoplethysmogram (PPG) signals for key management (see Section 1.5.3).

The next step was to specify the high-level design requirements and assumptions to build a low-level proposal. The following design requirements were considered:

- Provide a distributed platform to exchange and replicate EHRs
- Ensure unique patient identity across the platform
- Ensure unique EHR-to-patient mapping
- Ensure patient validation to EHR access control requests.
- Provide an access recovery mechanism for patients' EHRs
- Implement audit logging for access requests to EHRs
- Provide an EHR integrity validation mechanism
- Provide a mechanism for emergency access to EHRs by non-authorized providers
- Ensure the anonymity of EHRs in the datastore across the platform

Additionally, some assumptions were made that were excluded from the design requirements:

- All healthcare providers must use the SDHCARE system.
- All EHRs must be recorded in the same format (e.g., HL7 or FHIR).
- Secure links are used for communication between healthcare providers, the Blockchain and the datastore.

Based on the above requirements and assumptions, the low-level design proposal was developed. More details are provided in Chapter 2.

The next phase was a prototyping phase to build a sample that can implement the required functional and security features of SDHCARE. The low-level design blocks were analyzed, and the tools and software required to provide the desired features and functions for each design block were identified. The final selection of tools was based on the following criteria:

- Feature richness and simplicity of the tool.

- Security capabilities of the tool.
- Compatibility and interoperability between the tool and other components of the SDHCARE system.

The final phase of the research was testing and results analysis. This was divided into two sections:

- Validating that design requirements were achieved and can provide the expected functions.
- Performance evaluation to measure the average time and speed of transactions across the SDHCARE system.

The details of system prototyping, implementation, and testing are covered in Chapters 3 and 4.

## **1.5 Literature Review**

This section covers the literature related to blockchain technology and its adoption in EHR systems. The reviewed literature covers the progress of research in the following areas:

- Blockchain core functionality
- Blockchain security capabilities and limitations
- Limitations of legacy centralized EHR systems
- EHR security, privacy, and compliance requirements
- Existing proposals for Blockchain-based EHR systems

Table 2: Summary of the Literature

Category	Literature List
Blockchain Technology	Nakamoto [3], Buterin [4], Weber [8], Yaga, Mell, Roby, and Scarfone [9]
Blockchain Security	Joshi, Han, and Wang [7], Halpin and Piekarska [10], Zhong, Zhong, Mi, Zhang, and Xiang [12], Cash and Bassiouni [13], Tosh et al. [14]
Blockchain in Healthcare	McGhin, Choo, Liu, and He [5], Magyar [15], Pilkington [16], Zhang, Walker, White, Schmidt, and Lenz [17], Dagher, Mohler, Milojkovic, and Marella [18], Azaria, Ekblaw, Vieira, and Lippman [19], Xia, Sifah, Smahi, Amofa, and Zhang [20], Yang <i>et al.</i> [21], Roehrs, da Costa, and da Rosa Righi [22], Fan, Wang, Ren, Li, and Yang [23]

### 1.5.1 Blockchain Basics

According to the US National Institute of Standardization and Technology (NIST), Blockchain is defined as:

Distributed digital ledgers of cryptographically signed transactions that are grouped into blocks. Each block is cryptographically linked to the previous one (making it tamper evident) after validation and undergoing a consensus decision. As new blocks are added, older blocks become more difficult to modify (creating tamper resistance). New blocks are replicated across copies of the ledger within the network, and any conflicts are resolved automatically using established rules [6].

The entire blockchain is stored in each miner (as single unit) for synchronization instead of storing individual blocks. Figure 2 provides an overview of the structure of the blocks.

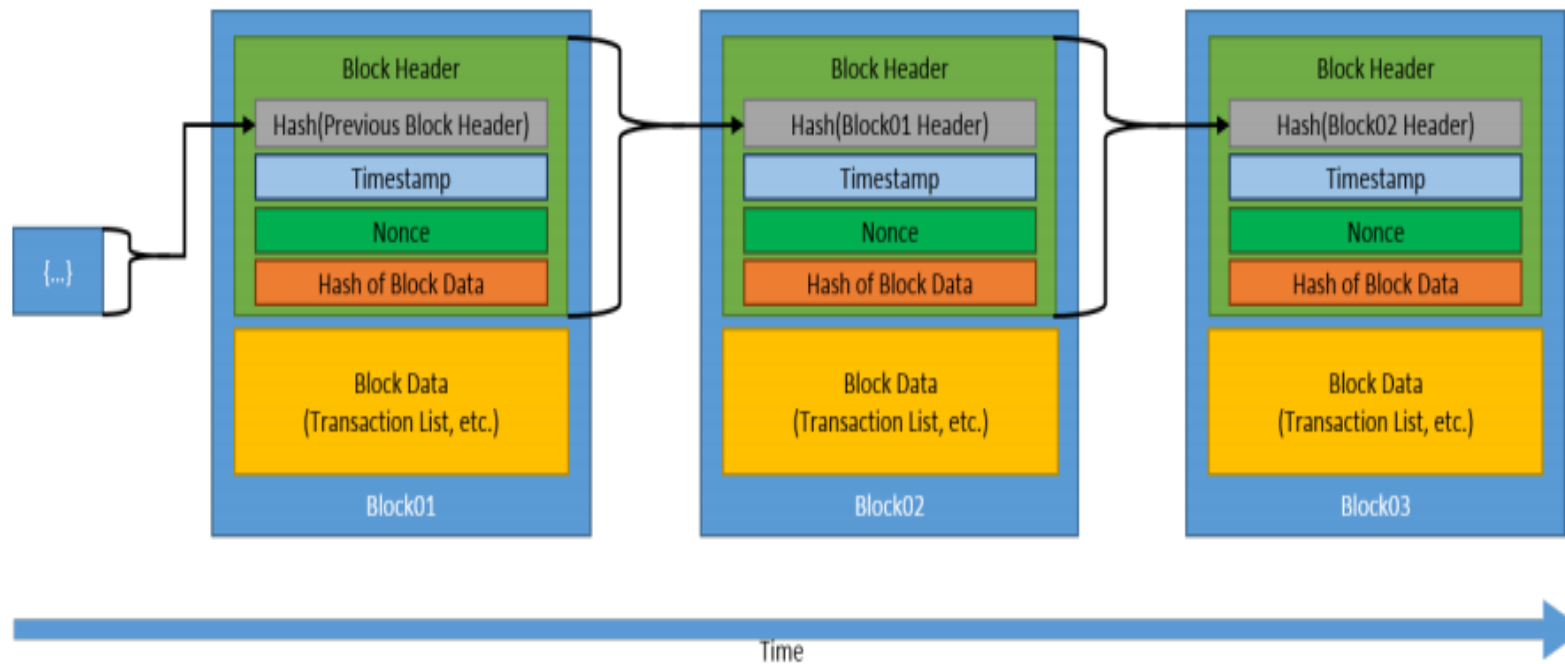


Figure 2: General Chain of Block



### 1.5.1.1 Block Components

Joshi, Han, and Wang [7] described the structure of the block as follows.

- **Data:** This is the application data held in the distributed blocks. The block can hold any type of data and is thus application-independent. Further, the block can hold multiple data units from diverse types of applications. Each data unit in the block is called a ‘message’ or a ‘transaction’.
- **Hash:** In a single block, three types of hash values exist: the hash value of the previous block (which is used to chain the blocks), the root hash representing all transactions stored in the block, and the hash value of the current block at the time it is committed to the chain.
- **Timestamp:** The timestamp at which the block was added to the chain.
- **Other information:** This includes information such as the software version used by the miner and the current difficulty level. An essential information element called a ‘nonce’ is used for block validation and the consensus algorithm. Figure 3 depicts this structure.

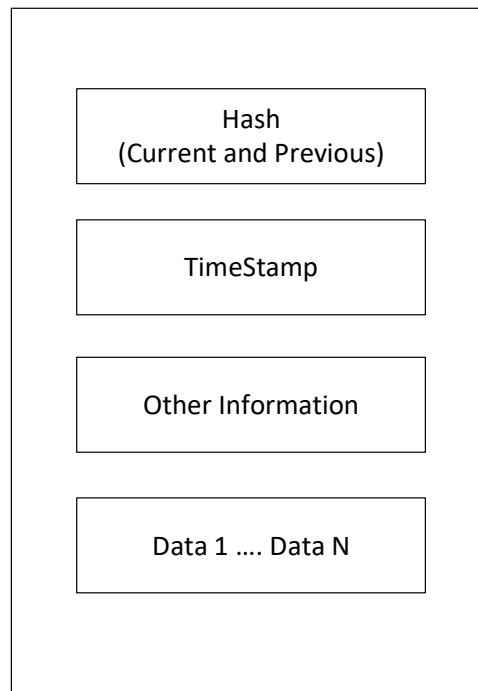


Figure 3: Block Structure

### 1.5.1.2 Types of Blockchains

Types of blockchains are defined according to how miners join the Blockchain network. Joshi, Han, and Wang [7] classified Blockchain networks as follows:

- **Public Blockchain (permissionless):** This type of Blockchain is publicly accessible without permissions or restrictions, eliminating the limitations of a central authority. Any node running the mining software (such as GETH for Ethereum Blockchain) can participate and start adding blocks, executing the consensus algorithm, voting to discard blocks, and obtaining access to any unencrypted information stored in the blocks.
- **Private Blockchain (permissioned):** This type of Blockchain is a permission-based platform established by a group of firms, individual firms, or divisions within a firm in which data can be accessed by users that are part of the mining group and properly authenticated.

- Consortium Blockchain: The consortium Blockchain is a hybrid between the no single trusted entity model of public Blockchain and the single, highly trustable entity model of private Blockchain. It is perceived as a partially decentralized Blockchain.

### **1.5.1.3 Transaction Lifecycle**

The lifecycle of transactions varies depending on the type of Blockchain. For example, in Ethereum, Weber et al. [8] summarized the transaction lifecycle as follows:

- The sender prepares its transaction with the application data and sends it to its local miner. The sender signs the transaction with its private key, which is validated by the local miner.
- The local miner generates a transaction ID and broadcasts the ID to the pool of miners. This transaction ID is a hash value of the hashed transaction.
- The miners maintain a pool of queued transactions. This queue is generally sorted based on the fees (called gas) paid by senders to process their transactions. Miners prefer to pick transactions with higher incentives (higher gas).
- Once a miner picks a certain number of queued transactions, it builds a block to include them. Next, the miner attempts to solve a crypto-puzzle to be elected to broadcast the block (This is called the consensus algorithm and varies based on the type of Blockchain. Ethereum uses a proof of work (POW) algorithm for consensus that is based on solving crypto-puzzles).
- Upon successful puzzle resolution, the miner posts the block in the Blockchain and waits for confirmation from other miners on the block that

it is committed to the main chain. Ethereum considers the presence of 12 proceeding blocks after the committed block as confirmation.

#### **1.5.1.4 Blockchain Consensus**

Yaga, Mell, Roby, and Scarfone [9] described consensus as the process that determines which user publishes the next block. Different models of consensus are used in Blockchains. These vary between CPU-intensive models, which is suitable for permissionless Blockchains for additional security, and low CPU models, which is suitable for permissioned Blockchains that assume a level of trust between miners. Some commonly-used consensus models are listed by [9]. The following sections describe the different models used in Blockchain for reaching consensus.

##### **1.5.1.4.1 Proof of Work (POW)**

In the POW model, a complex crypto-puzzle is published to all miners in the Blockchain. The first miner that solves the puzzle is granted permission to publish its blocks. The miner has to submit the solved puzzle as a “proof”, which is validated by other miners, and then the block is accepted. The difficulty of the puzzle varies and is continuously adjusted to maintain an average block committing time (e.g. 10 minutes in Bitcoin). Accordingly, POW is considered highly CPU-intensive and is usually used in permissionless Blockchains to reduce attackers’ interests in participating.

##### **1.5.1.4.2 Proof of Stack (POS)**

The POS model uses the amount of stack that miners invest in the system as an indication of their genuine intention and disinterest in compromising the Blockchain. The actual stack of the Blockchain varies based on its type, but cryptocurrency is generally used. The miners invest in the cryptocurrency of the Blockchain, which is

unusable expect for being a measure of their trust.

This model is currently being evaluated in permissionless Blockchains as it eliminates the high computational requirements of the POW model.

#### **1.5.1.4.3 Proof of Authority (POA)**

The POA model is based on maintaining a level of trust between miners and is used only in permissioned Blockchains. This level of trust is established through proven identities, which are verified by Blockchain members (e.g., authorized documents). During Blockchain runtime, the reputation of miners varies depending on their behavior, number of accepted blocks, and other factors. Miners with a better reputation will be awarded more slots to publish blocks while malicious miners will not be awarded sufficient slots to publish blocks.

#### **1.5.1.4.4 Round Robin Consensus**

Similar to the POA model, round robin consensus is limited to permissioned Blockchains. In this model, miners are awarded equal slots to submit blocks. An advantage of this model is that it guarantees that no miner can create a majority of blocks without the need for complex computation for validation.

### **1.5.2 Blockchain Security**

In this section confidentiality, integrity and availability (CIA) aspects of blockchain technology are investigated, as well as some common attacks against blockchain.

#### **1.5.2.1 Confidentiality**

According to NIST [6], confidentiality refers to “preserving authorized

restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information”. Hence, confidentiality is focused on authorization and privacy.

NIST [6] defines authorization as “access privileges granted to a user, program, or process or the act of granting those privileges”, which is not a feature natively provided by Blockchain technology. Similarly, privacy is not a native feature of Blockchain. Current implementations of Blockchain technology do not have the capability of encrypting the contents of the blocks or transactions before committing them to the ledger. The lack of authorization and privacy can introduce a substantial risk of a malicious miner joining the mining pool, obtaining a copy of the blocks and analyzing/exploiting their vulnerabilities. An example of this is the decentralized autonomous organization (DOA) attack [10]. DAO was an application developed and executed in Ethereum blockchain with the purpose of maintaining financial contributions for blockchain-based applications. It had an identity verification vulnerability which was exploited in 2016, leading to compromised security and losses totaling more than US\$50 million [11]. Lack of privacy and authorization allowed the attacker to read DOA blocks and identify the vulnerability.

Zhong, Zhong, Mi, Zhang, and Xiang [12] proposed a new model of privacy-protected Blockchain that encrypts data within an agreed upon time to add privacy capability to the technology. Cash and Bassiouni [13] proposed a two-tier Blockchain network that provides permissionless Blockchain at tier-1 with a POW consensus algorithm and permissioned Blockchain at tier-2 with a POA consensus algorithm. In this model, tier-2 provides access control to data specific to data owners and the users with whom they are sharing the data. Nodes from the permissionless tier can be

members of tier-2, which allows them to pass data and make transactions according to pre-defined access control contracts.

#### **1.5.2.2 Integrity**

NIST [6] defines integrity as “guarding against improper information modification or destruction and includes ensuring information non-repudiation and authenticity”. Nakamoto [3] explained how Blockchain technology implements strict integrity verification to avoid manipulation and tampering of data inside the blocks. This is implemented at three levels: the transaction, block, and miner level.

- At the transaction level, each miner verifies the hash value of the received transactions. Only valid transactions are queued in the block to be published in the ledger. The miner uses the sender’s public key to verify the received transaction hash value against the locally calculated value. Thus, this eliminates spoofing of senders’ public keys.
- At the block level, the miner uses the hash values of all transactions to calculate the root hash value that represents all transactions in the block (Bitcoin uses the Merkle tree while Ethereum uses the Patricia tree). The miner will use the hash value of the last block in the ledger, the root hash of the queued block, and the nonce value to calculate the local block hash. This nested hashing represents the actual chaining of the blocks and ensures blockchain integrity because tampering in any block will require a change in the hash values of all proceeding blocks in the ledger. Altering one block and all proceeding blocks is required before a new block is published in the ledger. Figure 4 below shows the detection of data tampering at the block level.

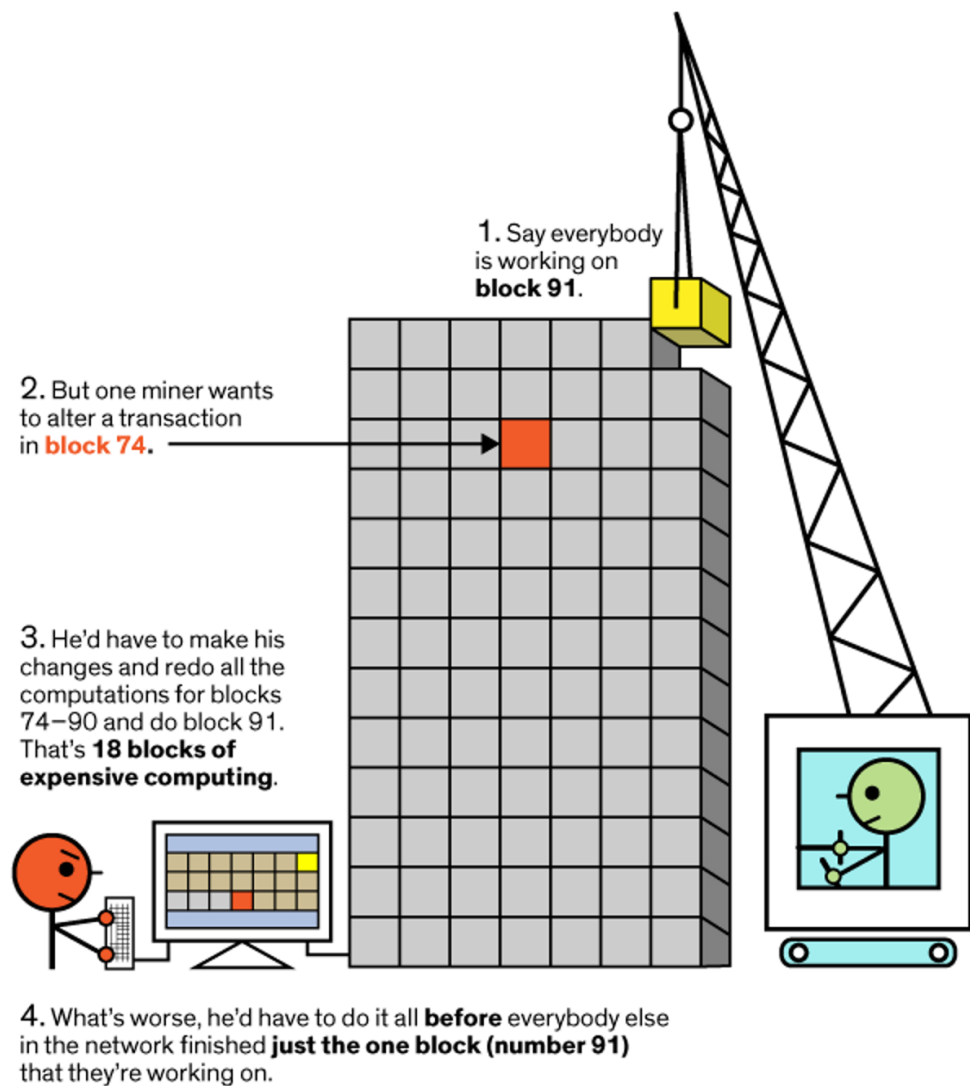


Figure 4: The Effects of Tampering with One Block in the Ledger

- At the miner level, for each new block, the miner verifies the block hash before accepting it in the ledger as the new last block. Further, the miner verifies the order of the new block by comparing the hash reference of the current last block in the ledger against the value listed in the new block. A block that fails the integrity check by miners or is placed out of order will be discarded if the majority of miners vote for the same result (i.e., more than 51% of miners vote to discard the block).



The nested hashing within the block and between blocks in the ledger guarantees the integrity of the data stored in Blockchain and makes it immutable (irreversible). This capability requires a thorough evaluation of the data policy before posting it on the Blockchain. If sensitive data such as medical records are encrypted and stored in the Blockchain and the encryption algorithm is later exploited, there could be significant exposure of sensitive information that cannot be revoked. Another example is a DOA attack where the Ethereum community cannot revoke the vulnerable DOA code without forking the ledger (i.e., discarding all the data inserted after the vulnerable DOA code).

#### **1.5.2.3 Availability**

Availability is defined by NIST [6] as “ensuring timely and reliable access to and use of information”. Availability is well implemented in Blockchain technology. The distributed architecture of the Blockchain network and the synchronization of the entire Blockchain across all miners in the pool provides robust resilience against single-point-of-failure scenarios. Failure in one or more node(s) does not stop the Blockchain network from introducing new blocks to the chain and/or serving access requests to the existing blocks. However, failure could impact the availability of the applications utilizing Blockchain.

Weber et al. [8] investigated the availability provided by Blockchain from an applications perspective. The research is focused on how the availability of Blockchain-based applications is affected by the time required to post a transaction on Blockchain. Using a sample of transactions from the Ethereum and Bitcoin Blockchains, Weber et al. found that 61.5% of transactions took more than 3 minutes to be committed in a Blockchain, while 13.8% of the transactions took more than 4.5

minutes to be committed. Such variation in time can introduce the unavailability of client applications.

Additionally, Weber et al. [8] investigated the factors that can influence the delay to commit transactions. Some of these are common between Ethereum and Bitcoin while others are unique to each, as indicated in Table 3.

Table 3: Factors for Blockchain Transaction Delay Variation

	<b>Ethereum</b>	<b>Bitcoin</b>
Transaction fees paid by the client	✓	✓
Transactions' order of arrival	✓	✓
Locktimes: indicating that a transaction is invalid until a certain block sequence number is mined		✓
Network delays	✓	✓
Gas limit (per block)	✓	

Weber et al. [8] analyzed the impact of DOS attacks on Ethereum Blockchain and found that the measures currently implemented by Ethereum that use a gas limit counter DOS attacks and provide strong availability for smart contract applications.

#### 1.5.2.4 Common Blockchain Attacks

Over time, different techniques have been used to attack active Blockchains, including Ethereum and Bitcoin. These attacks have targeted the core Blockchain technology rather than the applications running on the Blockchain (application attacks are subject to the vulnerabilities in applications). Tosh et al. [14] analyzed attacks on permissionless Blockchains that utilize POW as their consensus algorithm (the same attacks apply to permissioned Blockchains with lower risk due to the controlled

admission of miners). The following section reviews the literature on common attacks on Blockchain.

#### **1.5.2.4.1 Double-spend Attack**

In a double-spend attack, the same cryptocurrency (e.g., bitcoins) for more than one transaction. A malicious miner spends a certain number of bitcoins, for example, at block  $i$ . Starting from block  $i$ , the malicious miner privately mines to extend the Blockchain as fast as possible without publicizing it. The private chain does not include the bitcoins spent by the attacker in block  $i$ . Once the private chain (malicious) is longer (i.e., has more mined blocks) than the public chain (genuine), the malicious miner posts the malicious chain, which appears as a fork. Miners vote to resume the malicious chain and discard the genuine chain because the malicious chain is longer (which means that genuine transactions are discarded starting from block  $i$ , and the same bitcoins can be reused for other transactions).

#### **1.5.2.4.2 Selfish Mining Attack**

In a selfish mining attack, a pool of malicious miners colludes to mine together (which increases their hashing power) and distributes the incentives among themselves. This increases their profitability by enabling them to receive regular incentives (because they dominate the Blockchain with their hashing power) instead of receiving random incentives when mining independently. At the same time, it reduces the profitability of honest miners. Honest miners may prefer to join an honest pool to increase their mining power compared to a selfish pool, generating higher incentives. Thus, the pool eventually becomes the major player controlling the Blockchain, and decentralization no longer holds.

#### **1.5.2.4.3 Eclipse Attack**

An eclipse attack is aimed at isolating Blockchain miners and hijacking their mining power to form a malicious pool of miners controlled by the attacker. This malicious pool can be used to generate a fork and commit malicious blocks (i.e., overwrite the decentralization of Blockchain). Each miner in the ledger is limited to a certain number of concurrent, outgoing peers to maintain the peer-to-peer network (Bitcoin implementation limits each node to eight outgoing connection peers while Ethereum limits this to 11). In addition, it can accept a certain number of unsolicited incoming connections to form peers on the node's public IP (Bitcoin allows up to 117 connections).

Considering Bitcoin, each miner maintains a table of IPs for incoming peers sending unsolicited requests. The miner selects eight peers from the table to initiate outgoing connections. The selection of IPs is systematic (not random). An attacker can rapidly and repeatedly initiate unsolicited connection requests to the victim's node from attacker-controlled nodes. The victim's node populates its table with attacker-controlled miners' IPs. The attacker continues to overwhelm the victim's node with irrelevant information until it restarts. With such effort, there is a high chance that the victim will have the eight outgoing connections to attacker-controlled nodes after restarting, which isolates the victim from an honest pool, as indicated in Figure 5.

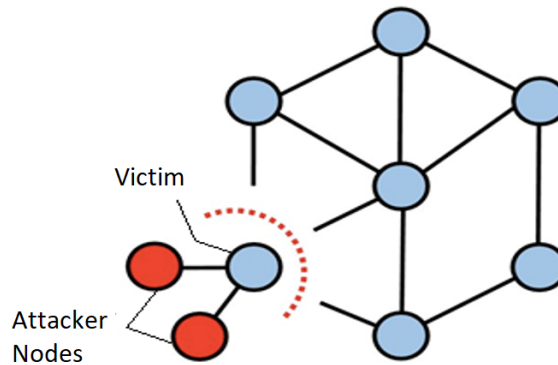


Figure 5: Eclipse Attack

#### 1.5.2.4.4 Block Withholding Attack

In this type of attack, some pool members who have joined to help mining blocks never publish any blocks, thus decreasing the expected revenue of the pool.

#### 1.5.2.4.5 Block Discarding Attack and Difficulty Raising Attack

Block discarding and difficulty raising attacks rely on the attacker's hashing power to mine blocks faster than others in the ledger, which leads to an increase of complexity, affecting the performance of the Blockchain network [14].

### 1.5.3 Blockchain in Healthcare

The literature reviewed in this section covers the challenges, capabilities, and some proposed implementations of blockchain in EHR, specifically:

- The challenges in integrating existing legacy centralized EHR systems
- The required capabilities in any blockchain-based EHR systems to be accepted
- Proposals for blockchain-based EHR systems, including:
  - MedRec

- BBDS
- OmniPHR
- MedShare

### **1.5.3.1 Centralized-Based EHR**

The move towards Blockchain-based EHR systems to integrate distributed healthcare providers raises questions about the challenges of integrating current centralized EHR systems that are distributed among healthcare providers with existing legacy technology. Magyar [15] listed four challenges in integrating existing centralized EHR systems:

- In centralized systems, EHRs are maintained in different formats that suit each provider's business model. This requires various interfaces and protocols for integration, and there is no single protocol accepted across all providers. Because of this complexity, there is high potential of compromising the security of EHRs and the privacy of patients across the different utilized middleware technologies and protocols.
- The current model of centralized systems provides high central authority to the dominant health provider of the patient, which complicates the exchange of health information in the case of unplanned treatment in an emergency situation and can cause serious results such as fatality because of the lack of timely access to EHRs.
- Auditing patients' history and traceability is a significant concern in centralized EHR systems as the information passes multiple healthcare providers. This is especially a concern when institutional incentives influence the history of a patient's data

- The availability of patient data in integrated, centralized EHR systems is inconsistent, and the related regulations are unclear. It is subject to the resilience of each centralized healthcare provider.

An example of integrating centralized EHR systems was a five-year agreement in 2016 between Google DeepMind and the Royal Free London NHS Foundation Trust. This integration encountered significant problems, which were summarized by Pilkington [16]:

- Lack of transparency and privacy
- Mismanagement of patients' data and identities
- Delayed treatment due to malicious software infections, which caused delayed service recovery

### **1.5.3.2 Blockchain-Based EHR Required Capabilities**

Zhang, Walker, White, Schmidt, and Lenz [17] conducted research on the required metric for any blockchain-based EHR system to be accepted. The researchers identified seven metrics:

1. Entire workflow of the system is HIPAA-compliant
2. Framework supports Turing-complete operations
3. Support for user identification and authentication
4. Support for structural interoperability at minimum
5. Scalability across large populations of healthcare participants
6. Cost-effectiveness
7. Support for patient-centered care model

#### **1.5.3.2.1 Entire workflow of the system is HIPAA-compliant**

For a healthcare solution to be accepted and adopted, it must fulfill the regulatory requirements of a country's National Health Authority (NHA). Considering the HIPAA Act, as an example of health regulation act in the US, Dagher, Mohler, Milojkovic, and Marella [18] analyzed its requirements and concluded that Title II of five HIPAA titles is relevant to Blockchain-based EHR. This title comprised the standards for privacy of individually identifiable health information (privacy rule) and the security standards for the protection of electronic protected health information (security rule). Magyar [15] further analyzed HIPAA requirements and concluded that Blockchain technology can fulfill the HIPAA requirements of secured access, privacy, lack of centralized government, and cost reduction.

Zhang, Walker, White, Schmidt, and Lenz [17] highlighted precautions that should be considered when implementing HIPAA-complaint Blockchain-based solutions. Peng et al. stated, "A core tenet of HIPAA compliance is that Personally Identifiable Information (PII) must be protected against a confidentiality breach. In particular, the end-to-end workflow of a healthcare app from entering to processing then delivering the data must be HIPAA compliant". This can be achieved in centralized systems using encryption techniques. However, in Blockchain, encryption may not be useful because any data stored in the Blockchain is replicated across all the miners and accessible by any party. Therefore, any breach of the currently used encryption algorithms makes the EHR information vulnerable, especially data in the Blockchain that is immutable and cannot be deleted. Accordingly, Zang et al. recommended storing encrypted metadata of the EHRs in the Blockchain (with a minimum level of information), which ensures that EHR data is securely stored [17].



#### **1.5.3.2.2 Framework supports Turing-complete operations**

Zhang, Walker, White, Schmidt, and Lenz [17] stated that any Blockchain-based EHR system should be Turing-complete and have programming capabilities to enable simple integration and interoperability with legacy systems. In addition, it should have the capability for simple upgrades and feature enhancements. Blockchain networks built specifically for healthcare applications are not scalable and cannot fulfill these requirements.

#### **1.5.3.2.3 Support for User Identification and Authentication**

In EHR systems, users are classified as patients and healthcare professionals. As [17] stated that any Blockchain-based EHR system should be able to uniquely identify and distinguish each user while maintaining their anonymity on the Blockchain, securely authenticate users, and be capable of recovering user's authentication information if it is lost or stolen.

#### **1.5.3.2.4 Support for Structural Interoperability at Minimum**

The system should enable the exchange of medical data and interpretation of received data in its current standards [17], i.e., the system should be able to communicate with known industry standards such as FHIR and HL7.

#### **1.5.3.2.5 Scalability across Large Populations of Healthcare Participants**

This was described in [17] thus: "A successful health app should leverage the Blockchain to enhance interoperability, while maintaining its quality when users or components of the app scale up and out".

#### **1.5.3.2.6 Cost-effectiveness**

Any blockchain-based EHR system should be cost-effective compared to the existing legacy systems without affecting its capabilities [17]. This factor has a significant impact on the selection of blockchain parameters, including type, consensus algorithm and incentives model.

#### **1.5.3.2.7 Support of Patient-centered Care Model**

According to [17], any Blockchain-based EHR system should provide patients with the ability to control or monitor their information without compromising other functionalities. These features may include self-reporting health information, access to personal medical records and prescription history from different providers, auditing existing access to patient health records, and the ability to share or revoke access to patients' own medical data.

#### **1.5.3.3 MedRec**

MedRec was proposed by Azaria, Ekblaw, Vieira, and Lippman [19] to utilize Blockchain technology to integrate existing centralized EHR systems among distributed healthcare providers. The solution uses Ethereum Blockchain's smart contracts capability to facilitate this integration. Each healthcare provider should contribute an Ethereum mining node (usually a dedicated server) to participate in MedRec. In addition, patients should also contribute an Ethereum mining node (on a PC or mobile device) to participate in MedRec. The main functions of MedRec are to:

- Enable inter-provider access to patients' EHRs using API interfaces. The API information of the providers is stored in the Blockchain.
- Provide patients with the capability to manage access control to their EHRs.

Access control lists for patients and providers are stored in the Blockchain.

- Detect and notify patients about new access requests to their EHRs. Access can be granted or rejected only by patients.
- Notify patients about changes to their EHRs and log the changes in the Blockchain.
- Provide a copy of EHRs on patients' nodes and dominant providers' nodes.

Although MedRec accelerates the deployment of EHR systems by integrating with existing systems using Blockchain technology to overcome the major limitations of centralized EHR systems, MedRec has shortcomings that limit its feasible production implementation:

- A mandatory component of MedRec is the presence of patients' nodes, which are used to communicate with patients for access control management. This limits the scope of MedRec solutions to Blockchain-enabled patients (i.e., patients should have an Ethereum account). This is a major limitation of the solution from the patients' perspective (but not the providers' perspective). Any proposed solution should be capable of supporting all patients without restrictions.
- Ethereum Blockchain uses POW as its consensus algorithm, which is known to have significant computing power requirements. While healthcare providers can contribute powerful mining nodes to use MedRec, this cannot be (practically) achieved for patients, whose nodes are on PCs or mobile devices, making MedRec practically infeasible.
- If a patient loses the private key to their Ethereum account (which is possible using a mobile device or PC), MedRec does not provide a mechanism for a

patient to recover control of their EHR.

- The use of current centralized EHR systems raises an interoperability problem regarding inter-provider access. The solution must assume that all providers utilize the same EHR format standard such as HL7 or FHIR [19], which is not the case with the current centralized systems.
- MedRec does not provide a mechanism for emergency access to EHRs if a patient is admitted to a non-authorized hospital for emergency treatment.

#### **1.5.3.4 BBDS**

Xia, Sifah, Smahi, Amofa, and Zhang [20] proposed a Blockchain-based data sharing (BBDS) system to provide access control management to EHRs stored in the cloud based on the Blockchain technology. The proposed BBDS system utilizes permissioned Blockchain consisting of an issuer that grants users or organizations access to the system, a verifier that validates requests from system members and grants corresponding access rights, and consensus nodes that facilitate the interface between members and the verifier in addition to logging requests in Blockchain for auditing and forensics purposes. The BBDS system provides the following functionalities:

- A proof of verification (POV) algorithm between the issuer and users/organizations to enroll them in the BBDS system. The POV algorithm is based on a proposed lightweight Diffie-Helman key exchange to generate a session key for encryption and an electronic registration form to be validated by the issuer.
- Controlled access to EHRs, stored in the cloud, by a verifier node for members of the BBDS system. This verification process is based on a per-member private key generated during the registration phase by the issuer and

communicated to members and the verifier. After successful verification of a member's identity, the verifier validates the request against member rights and, if access is granted, the verifier retrieves data from the cloud and passes it to the member, or read data from the member and posts it in the cloud.

- Audit logging in the Blockchain ledger using consensus nodes, where each member's request to read or post an EHR is stored in a separate block. The information recorded in the block includes user identity, purpose of the request, processing consensus node, verification result, and timestamps, including request creation, request retrieval from unprocessed requests pool, verification time, block broadcast time, and data send time.

Although the BBDS system is not limited by cryptography key recovery and emergency access restrictions (because the actual EHRs are not stored in the Blockchain and their access is controlled by the issuer/verifier), it has other limitations with the current implementation:

- The use of permissioned Blockchain eliminates decentralized authority, which is a core advantage of Blockchain technology. The model provides central authority to the issuer (not the patient) to verify and accept members in the system.
- Because of the use of non-Turing-complete Blockchain (i.e., no smart contracts), the BBDS system records each event in a single block to be able to uniquely identify the events by the block reference. This limits the scalability of the system because of recording a large number of blocks in a very short time.
- The proposed model provides the data to the requester before recording the

request details in the Blockchain, which introduces a vulnerability in the system as data is provided without a recorded request.

- The proposed BBDS system does not have a mechanism to detect modifications/tampering in EHRs in the cloud caused by system-independent reasons such as malicious activity in the cloud.

### 1.5.3.5 MedShare

MedShare was proposed by Yang et al. [21] to connect centralized healthcare entities and exchange EHRs using a hybrid cloud infrastructure. The proposal was prototyped with three healthcare entities: Hospital Conde S. Januário (HC), Kiang Wu Hospital (KW), and Macau University of Science and Technology Hospital (UH). Medshare functions as follows:

- Each healthcare entity has a private cloud that converts EHRs from the entity's specific format to a standard EHR format and stores them locally in the private cloud. In other words, each entity has two copies of an EHR, in a standard format and a non-standard format.
- Standard format EHRs are indexed using hash maps, and the index values are stored in a public cloud that is connected with private clouds. The public cloud has a synchronizer component that is used to replicate per-patient EHRs across all private clouds (scheduled replication).
- Doctors locally authenticate with the healthcare entity and query the EHRs of patients. If a healthcare entity cannot find a patient ID locally (assuming replication not yet been done), it queries the public cloud to locate the patient's EHR and, after successful validation, obtains the EHR.

The Medshare model is very practical and overcomes major limitations in

legacy EHR systems, but it lacks the following:

- Neither private nor public clouds guarantee immutable access control rules, privacy isolation between patients, or immutable integrity verification, which is provided by Blockchain technology.
- The replication of EHRs between healthcare entities is not scalable when a large number of healthcare entities are involved in the system. This requires  $n \times (n - 1)$  connections to achieve full replication of EHRs.
- Medshare uses patients' ID cards as a mechanism for uniquely identifying patients and obtaining their consent to grant healthcare providers access rights to their EHRs, which is known to be an insecure technique compared to biometrics identity verification.

#### 1.5.3.6 OmniPHR

Roehrs, da Costa, and da Rosa Righi [22] proposed a Blockchain-based application, OmniPHR, to address the following problems:

- Provide a unified view to patients of their healthcare records from anywhere at any time.
- Provide up-to-date information to healthcare providers about patients regardless of whether the data is local to the provider or is from an external provider.
- Provide a single standard for healthcare records.

Each member of OmniPHR joins the Blockchain through a miner, called a leaf node. OmniPHR uses a 'routing overlay' node (called a super node), which is responsible for managing leaf nodes and inter-communication with other routing

overlays. Some other roles of the routing overlay node are:

- EHR handling: Accepting input medical records from IoT devices or healthcare organizations, converting them to an open EHR format (the standard EHR format used by OmniPHR), dividing the EHRs into chunks of blocks, and distributing the blocks across Blockchain miners using load-balancing algorithms
- Security: Encrypting blocks, signing blocks, validating blocks, and providing access authentication and access control to the blocks.

OmniPHR has limitations in its capability to provide a unified EHR system:

- Similar to BBDS, the use of non-Turing-complete Blockchain adds significant complexity for additional features or enhancements to the system compared to Turing-complete Blockchain, which can have added features through software coding.
- Storing large data in the Blockchain (e.g., X-rays and MRI scans) is not practical due to the size requirements on the nodes, which entails significant overhead in addition to the encryption and decryption processing overhead.
- The proposed model does not uniquely identify the author of data because all the blocks are signed by the leaf nodes or super nodes.
- Access to EHRs should be authorized by patients, which does not address the limitation of unplanned treatment such as emergency admission by unauthorized healthcare providers.
- OmniPHR does not overcome the limitation of duplicate data such as duplicate patient registration information that occurs when a patient registers at two healthcare providers with different identities. Ideally, OmniPHR should have



a mechanism to uniquely identify each patient without duplication, such as biometric identity.

#### **1.5.3.7 MedBlock**

Fan, Wang, Ren, Li, and Yang [23] proposed the MedBlock system to share medical data efficiently using Blockchain. The MedBlock system generates a private/public key pair for each patient that is used to encrypt and sign medical records. The actual records are stored in the health provider's local database while the Blockchain holds the hash value of the records. The core functions provided by MedBlock are:

- A dedicated certificate authority server is used to generate keypairs for patients, community hospitals and national hospitals.
- Patients submit their records through community hospitals or national hospitals signed with their private key and encrypted by their public key.
- Health records are not stored in community hospitals. Instead, they are stored directly in national hospitals' databases.
- The department that accepted records from a patient signs them using its local private key to ensure integrity and non-repudiation.
- Each geo-group of national hospitals has the same group of endorsers that will build the blocks and submit them to the consensus nodes (called orderers). The hospitals submit the hash value of the medical records to the endorsers, and the records are stored in the local database.
- Once the orderers reach consensus, they post the block to the ledger.
- Access control is implemented by MedBlock through private key

signatures. The client application scans the blocks until a valid signature is found that corresponds to the patient's data.

While MedBlock provides an efficient and scalable mechanism using role-based nodes to perform specific functions and guarantee security through double-signing, it has the following drawbacks:

- The use of private/public keys for patients to sign and encrypt records creates an issue in the case of unplanned treatment such as emergency admission. In such a case, medical records will not be accessible, which can cause complications with treatment and even fatality.
- If a private key is lost, patients cannot recover their medical records.
- The access control mechanism used is inefficient, especially when the ledger grows to a very large number of blocks. In this case, examining all the blocks until the records are found is not scalable.
- The lack of programmability is a major drawback of MedBlock if new features are required, as this would require the addition of new nodes.
- MedBlock does not provide a mechanism to exchange medical records between hospitals as the records are stored in the local database of national hospitals.

## **Chapter 2: Methods**

This chapter addresses the contribution to the problem statement and proposes a solution to ensure an access recovery mechanism for patients' EHRs that are exchanged and synchronized between distributed healthcare providers using Blockchain. The high-level architecture of the solution is discussed, followed by a detailed explanation of individual components and layers. In addition, functional use cases of the design are provided and explained.

### **2.1 Design Overview**

The approach followed in our solution is to divide the system into layers and provide distributed functions in a modular structure. Accordingly, the system is divided into four layers, namely the User Interface, Middleware, Blockchain, and Cloud Store. Figure 6 shows the layers that comprise SDHCARE and the modules associated with each layer.

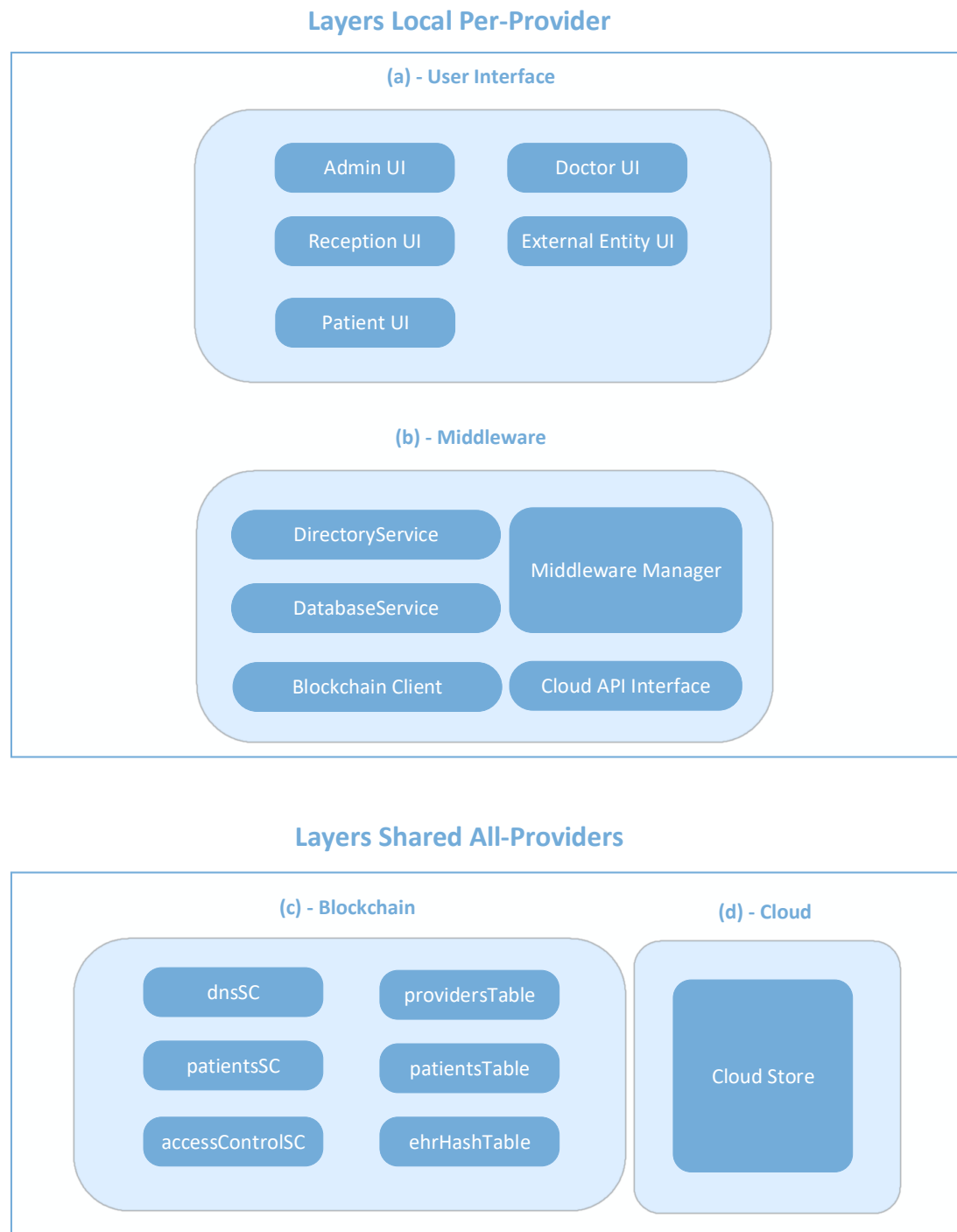


Figure 6: SDHCARE High-Level Architecture

### 2.1.1 User Interface (UI) Layer

The UI layer is the presentation of the system to the users, by which they can interact with SDHCARE. These users can be doctors, pharmacists, receptionists,

officers, researchers, or insurance companies, among others. This layer is local to each healthcare provider that is a member of SDHCARE. It has mandatory and customized components according to a provider's needs. The mandatory components are summarized as:

- Admin UI: This interface is mainly used by authorized representatives of the healthcare provider to enroll the provider in the SDHCARE system. Upon successful enrollment, the provider's details are stored in a providers' immutable table for it to be uniquely identified in the SDHCARE system.
- Reception UI: This interface is used by a healthcare provider's reception department. It provides two main functions:
  - Registering new patients in the system, which involves creating an entry for each patient in an immutable-blockchain table that includes all patients in SDHCARE. This ensures exchange and synchronization of all patients across SDHCARE healthcare providers.
  - Booking and confirming patients' appointments, which grant the corresponding clinics and/or departments access to patient's records (after the patient's confirmation using his/her fingerprints). This access is stored and maintained along with the patient's registration record.
- Patient UI: This is a biometric interface that accepts patients' fingerprints and uses them for the following functions:
  - Uniquely identify patients using fingerprints instead of secret keys (or private keys) across all members of SDHCARE.

- Managing access and role assignments by patients using fingerprints, which eliminates the problem of recovering a patient's management of access to their records in the case of a lost secret key
- Indexing patients' records metadata stored in the blockchain records table
- Simplify the process of granting access to patient records in the case of emergency treatment using fingerprints
- Doctor UI: This interface is used by the doctors to view patients' EHRs and post new EHRs after successful validation of a patient's identity and authorization rules implemented in the SDHCARE system.
- External Entity UI: This interface enables remote healthcare providers, research entities, insurance companies, and other parties to obtain access to patient records after successful validation of access requests to provide globalized access to the SDHCARE system.

### **2.1.2 Middleware Layer**

This layer is the core of the SDHCARE solution. It is used to interlink all of the other SDHCARE layers in addition to providing major SDHCARE services, including directory services and database services. Figure 6 (b) lists all modules included in the Middleware layer. These modules are local to each healthcare provider participating in the SDHCARE system and can be summarized as follows:

- DirectoryService Module: This module is mainly used to host the directory of all accounts of the healthcare provider. Local identities within the healthcare provider (e.g., doctors, nurses, system admins,

receptionists) are validated against the DirectoryService module. Additionally, it is responsible for access control validation on UIs.

- DatabaseService Module: This module is responsible for holding all database information local to the healthcare provider, including clinics' IDs, provider registration details, and patients' appointments.
- Middleware Manager Module: This module acts as the main controller for interacting between the modules within Middleware layer as well as intercommunication with other layers including UI, Blockchain and Cloud Store layers.
- Cloud Application Programmable Interface (API) Interface Module: This module is mainly used to facilitate communication between the Middleware Manager module and the Cloud Store to read or post healthcare records.
- Blockchain Client Module: This module is responsible for the communication between the healthcare providers and Blockchain layer. All interactions between the Middleware Manager module and the Blockchain layer go through this module. Two types of operations exist within this module:
  - Write requests to publish new blockchain transactions of data to the Blockchain smart contracts and immutable tables.
  - Read requests in the form of data transactions consolidated in blocks to read data from the immutable tables or responses from smart contracts.

### 2.1.3 Blockchain Layer

The Blockchain layer is a shared layer across all healthcare providers participating in SDHCARE. This layer provides two core functions in the SDHCARE system, namely:

- Immutable smart contracts (SC) to perform programmed logic functions.
- Immutable tables in the form of chained blocks to store different types of data that need to be protected against unauthorized tampering.

The design of the immutable tables is distributed to ensure the anonymity of the data stored in these tables. For example, the visibility of `ehrHashTable` should not provide any correlation to the patients in `patientsTable`. This correlation is controlled by the middleware layer using patient's fingerprint hash. Similarly, the design of the smart contracts is distributed to ensure modularity and flexibility for additional features and enhancements.

Figure 6 (c) summarizes the components of the Blockchain layer. The next sections provide details of each component.

#### 2.1.3.1 DNS Smart Contract (`dnsSC`)

The `dnsSC` is programmed to perform the initial enrollment of the healthcare provider in the SDHCARE system. It takes the provider's information from the Admin UI and stores it in `providersTable`. In addition, it ensures the uniqueness of the provider's information across all the members of the SDHCARE system.



### 2.1.3.2 Patients Smart Contract (patientsSC)

This smart contract plays a vital role in the SDHCARE system and is responsible for multiple functions:

- Initial registration of patients in the SDHCARE system, which includes populating patients' details in patientsTable.
- Updating ehrHashTable with new EHR metadata and their corresponding hash values.
- Retrieving patients' details from patientsTable.
- Retrieving the list of EHRs from ehrHashTable.

### 2.1.3.3 Providers Table (providersTable)

providersTable is an immutable table stored in Blockchain (in the form of chained blocks) and holds information about all healthcare providers that are members of SDHCARE. It is populated using dnsSC. Figure 7 outlines the information held in providersTable:

- Name: Healthcare provider's name
- Web address: Healthcare provider's domain name
- OHP: Healthcare provider's Blockchain address

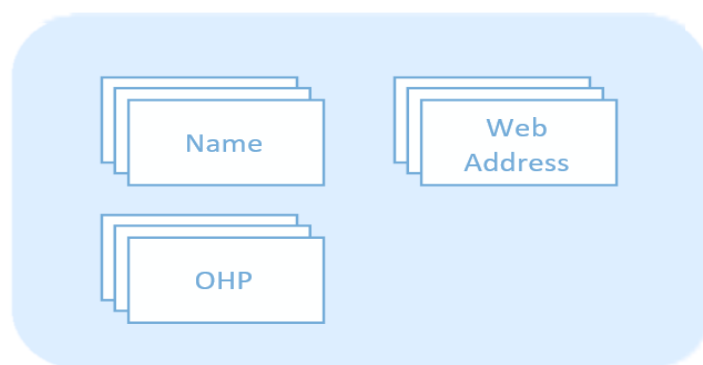


Figure 7: Summary of providersTable Data

### 2.1.3.4 Patients Table (patientsTable)

This immutable table stores information about patients and is populated using patientsSC. The data in patientsTable is indexed using patients' fingerprints hashes, which is critical for the unique identification of patients. In addition, the fingerprints hashes indicate the relationship between patientsTable and ehrHashTable. Figure 8 summarizes the data stored in this table:

- Fingerprint Hash: Patient fingerprint hash
- OHP: This is used to trace the origin of the patient record
- PACL: This is the patient access control list array and includes the IDs of clinics allowed to access a patient's EHR
- Receptionist ID: The ID of a local's provider reception department is used for auditing purposes

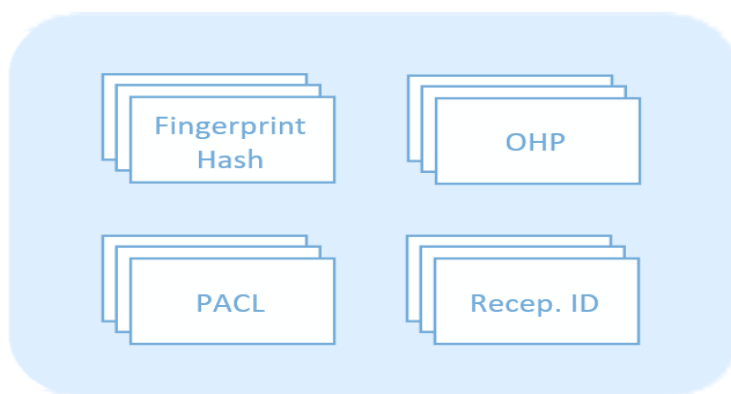


Figure 8: Summary of patientsTable Data

### 2.1.3.5 EHR Metadata Table (ehrHashTable)

The EHR Metadata Table holds the metadata of patients' EHRs and their corresponding hash values. These hash values are used for indexing EHR raw data uploaded to the cloud store. Additionally, these hash values are used for detecting any unauthorized changes to EHRs. Similar to patientsTable, ehrHashTable is indexed

using patients' fingerprints hashes to ensure unique mapping between patients' EHRs and their identities.

Our design stores the hash values of the EHRs in ehrHashTable instead of the EHRs raw data for two reasons:

- In Blockchain, blocks are not encrypted, accordingly, storing clear-text EHRs breaches the privacy of patients. Further, storing encrypted EHRs makes them subject to crypto-analytic attacks, and any success in cracking the encrypted algorithms would breach patients' privacy.
- Blocks are stored locally on the miners. This method can be used for small EHRs. However, for large EHRs such as MRI scans and X-ray images, the solution won't be scalable.

Figure 9 summarizes the information included in ehrHashTable:

- EHR Name: EHR name
- EHR Date: EHR creation date
- EHR Hash: EHR Merkle root hash
- Fingerprint Hash: Patient fingerprint hash
- EHR Status: EHR status (active/deleted)

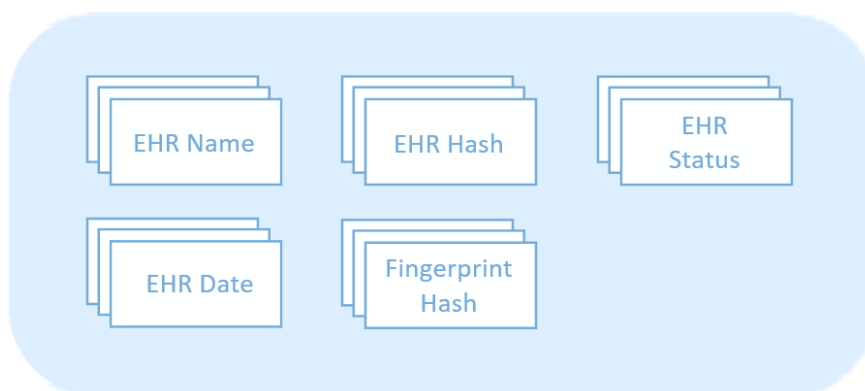


Figure 9: Summary of ehrHashTable Data

The hash value of the EHR is calculated using a Merkle Tree algorithm as follow:

$$H1 = \text{SHA256 (EHR Name)} + \text{SHA256 (EHR Raw Data)}$$

$$H2 = \text{SHA256 (Date)} + \text{SHA256 (EHR Status)}$$

$$\text{Merkle Root Hash} = \text{SHA256}(\text{SHA256 (H1)} + \text{SHA256 (H2)})$$

Finally, the Merkle root hash is used as an indexing key in the cloud store to look up EHRs raw data.

#### 2.1.3.6 Access Control Smart Contract (accessControlSC)

The accessControlSC acts as access control manager for any activity on patients' EHRs. It validates all requests including:

- Writing EHR records in ehrHashTable.
- Reading EHR records from ehrHashTable.
- Granting health provider clinics access to EHRs.

Table 4 summarizes the access control matrix provided by accessControlSC.

Table 4: Access Control Matrix by accessControlSC

	<b>providersTable</b>	<b>patientsTable</b>	<b>ehrHashTable</b>
<b>Reception</b>		Read Write	
<b>Doctor</b>		Read	Read Write
<b>Patient</b>		Grant	Grant
<b>Officer</b>	Read/Write		

#### **2.1.4 Cloud Store Layer**

The Cloud Store layer is a shared layer across all healthcare providers participating in the SDHCARE system. It holds patients' raw EHRs, which are indexed using Merkle root hash values of the EHRs. This makes EHRs in the cloud store completely anonymous. In other words, an EHR cannot be traced back to a patient's identity because patient's identity, fingerprint, is not stored in the cloud store, blockchain or providers' DatabaseService modules.

### **2.2 Functional Use Cases**

This section explains SDHCARE operation for major use cases that exist in any healthcare provider deploying SDHCARE. These use cases include provider enrollment in SDHCARE, new patient registration, patient appointment management, and doctors' access to patients' EHRs.

#### **2.2.1 SDHCARE Provider Enrollment**

For enrollment in the SDHCARE system, a provider should select a unique name and web address for global reachability. The next step is storing this information in providersTable. This helps other providers verify the uniqueness of their information before enrollment in SDHCARE. Further, it is used by all SDHCARE healthcare providers to identify the source of patient EHRs. The process of provider enrollment, shown in Figure 10, is summarized as follow:

1. From the Admin UI, the provider's admin is authenticated against the provider's DirectoryService.

2. After successful authentication, the admin enters the provider's details in the Admin UI (name and web address).
3. The provider's Middleware Manager sends a request to the Blockchain through its Blockchain Client to create a new account, which entails generating a keypair. The private key is used to sign all requests (transactions) from the provider. The public key is used as the provider's Blockchain address to uniquely identify it across the ledger (OHP). This keypair is returned to the provider from the Blockchain and stored in DatabaseService component of SDHCARE
4. Next, the Middleware Manager obtains the provider's OHP from the DatabaseService component and the Blockchain Client calls the dnsSC to update providersTable with the provider's name, web address, and OHP.

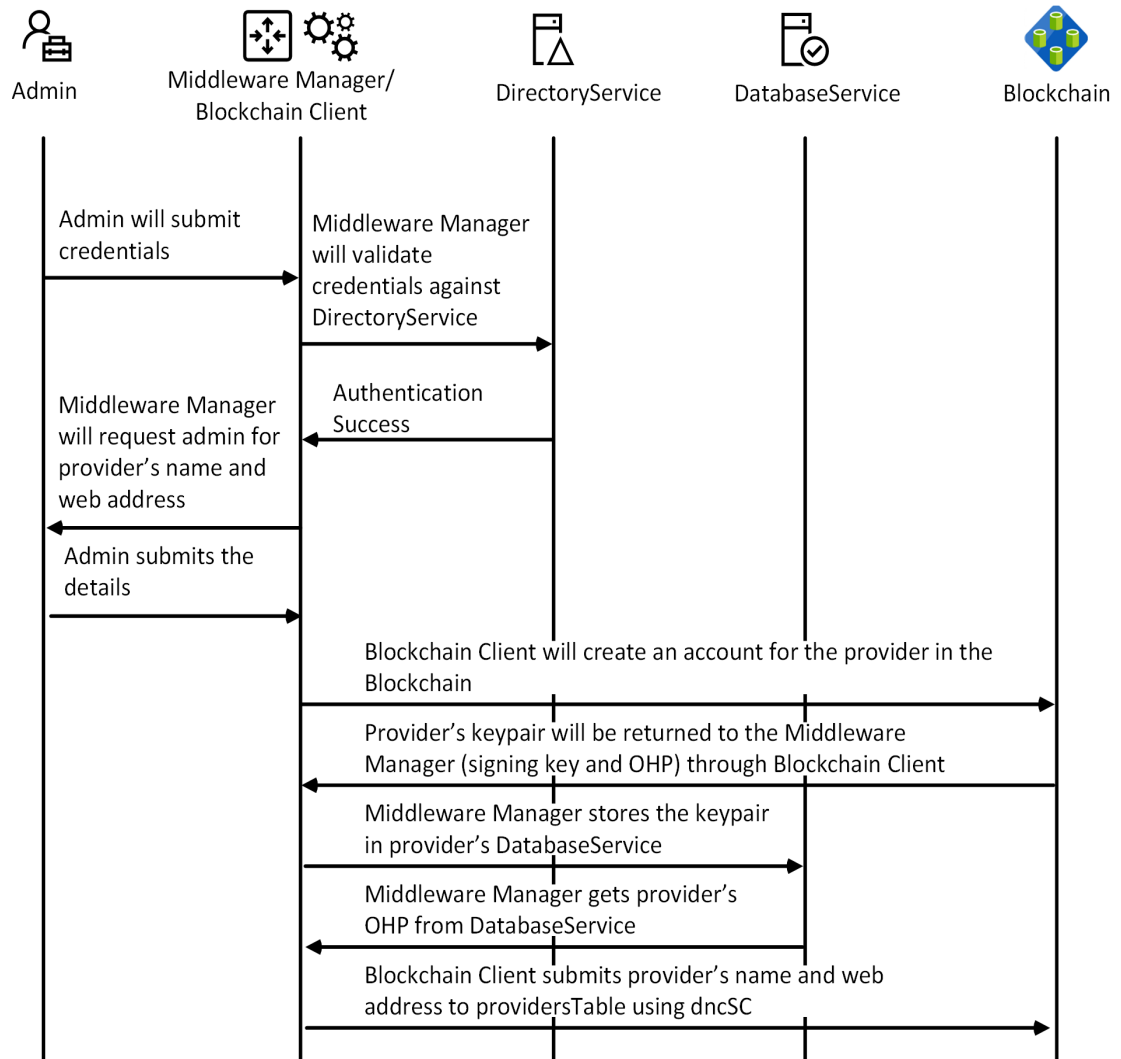


Figure 10: Hospital Enrollment Sequence Diagram

### 2.2.2 Patient Registration

New patients visiting a SDHCARE-enabled healthcare provider go through the registration process to be added to patientsTable. Once patients complete the registration process, they are uniquely identified across all SDHCARE-enabled providers. This unique identification is achieved using patients' fingerprints hashes for patientsTable indexing. The process of adding new patients to the SDHCARE system is described below and summarized in Figure 11.

1. The receptionist uses Reception UI to authenticate against a provider's DirectoryService.
2. After successful authentication, the receptionist captures the patient's details that are required by patientsTable through Reception UI.
3. The patient validates and confirms the details by submitting his/her fingerprint through Patient UI. A fingerprint is a mandatory input to resume the process.
4. Next, the provider's Middleware Manager obtains the provider's OHP and the reception ID from the provider's DatabaseService. This information is passed to the Blockchain Client.
5. Finally, the provider's Blockchain Client calls patientsSC to store the patient's details entered by the receptionist, along with the provider's OHP and reception ID, in patientsTable indexed using the patient's fingerprint hash.



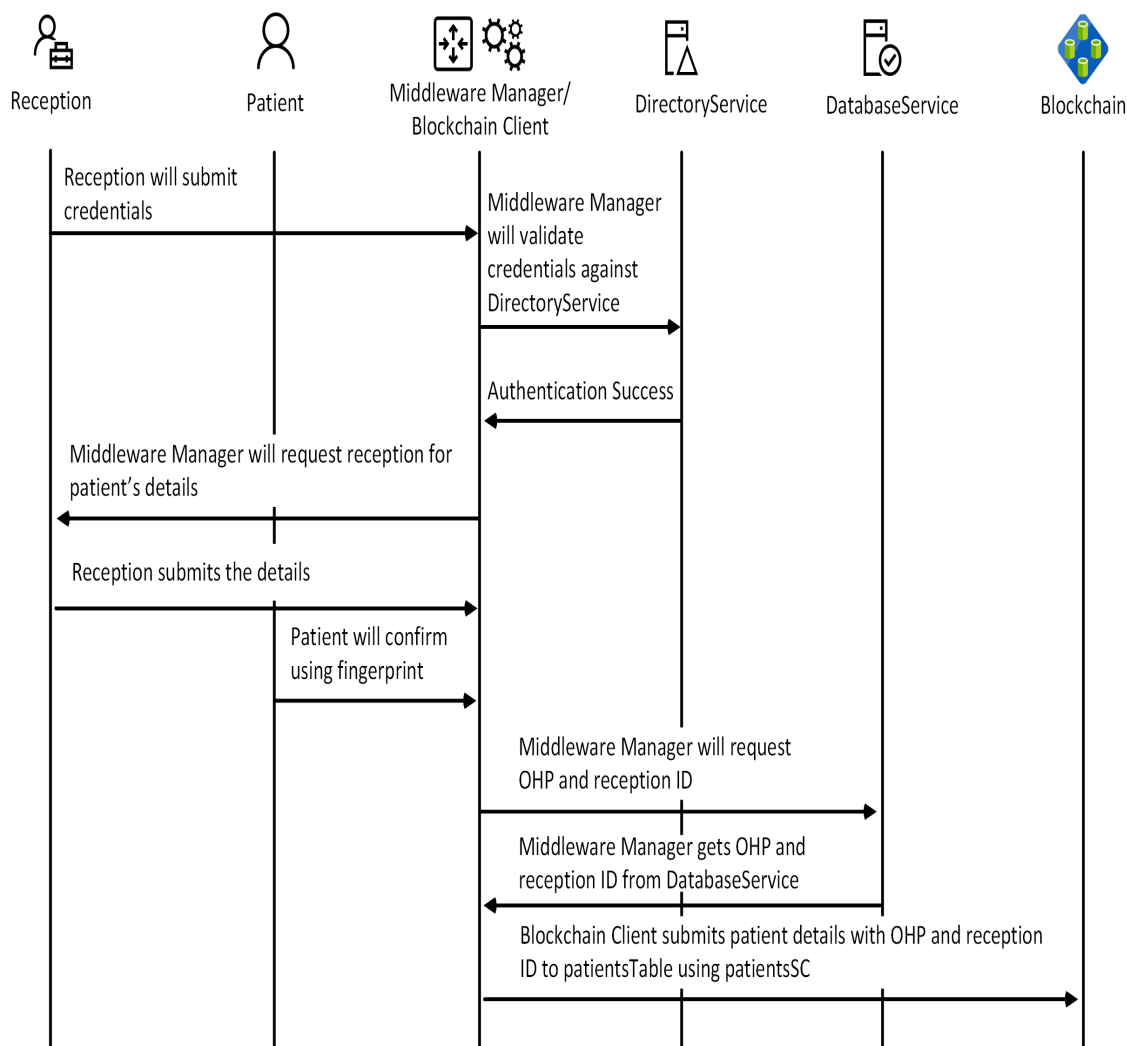


Figure 11: Summary of the Patient Registration Process

### 2.2.3 Patient Appointment Management

SDHCARE-reception manages patients' appointments, including booking new appointments and acknowledging existing appointments attended by patients. The appointments are stored in each provider's DatabaseService and are indexed using patients' national IDs (NIDs). Additionally, the appointments are not synchronized between providers in the SDHCARE system. A patient attending an appointment needs to confirm the attendance by submitting his/her fingerprint. This confirmation grants

the clinic access to the patient's EHR. Figure 12 summarizes the process of booking a new patient appointment, which entails the following steps:

1. The receptionist uses Reception UI to authenticate against the provider's DirectoryService.
2. After successful authentication, the receptionist provides appointment details, including the patient's name, NID, date/time, and clinic.
3. The Middleware Manager receives the details of the appointment from the Reception UI and stores them in the DatabaseService of the local provider.

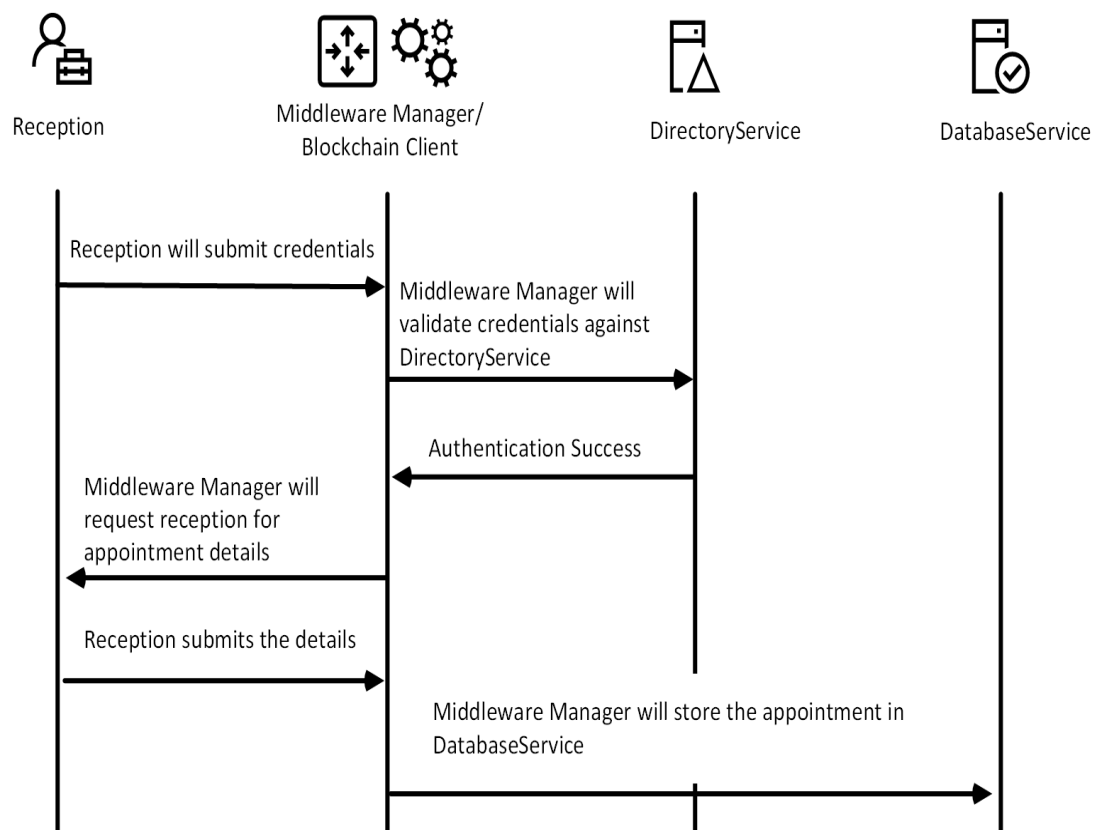


Figure 12: Process for Booking a New Appointment

The process of confirming a patient's appointment and granting access to the visited clinic is described below and summarized in Figure 13:

1. The receptionist uses Reception UI to authenticate against the provider's DirectoryService.
2. After successful authentication, the receptionist looks up the patient's NID against DatabaseService to obtain a list of the patient's active appointments.
3. The receptionist selects the desired appointment and the patient confirms it by submitting his/her fingerprint using Patient UI.
4. Upon confirmation, the Middleware Manager queries DatabaseService to obtain the visited clinic's ID and deactivate the selected appointment.
5. Finally, the Middleware Manager calls accessControlSC through the Blockchain Client to append the clinic's ID to the PACL stored in patientsTable and indexed using the patient's fingerprint hash. This grants the visited clinic access to the patient's EHR.

It is important to note that the Blockchain Client uses the clinic ID + OHP as the format of the clinic ID stored in PACL. This eliminates the possibility of conflict caused by providers having the same clinic IDs and being granted access to the same EHRs. For example, if the clinic ID is 100 and the OHP is 1000000000000000000001, the clinic ID stored in PACL is 10010000000000000000000001, which is guaranteed to be unique due to the uniqueness of the OHP in the Blockchain.

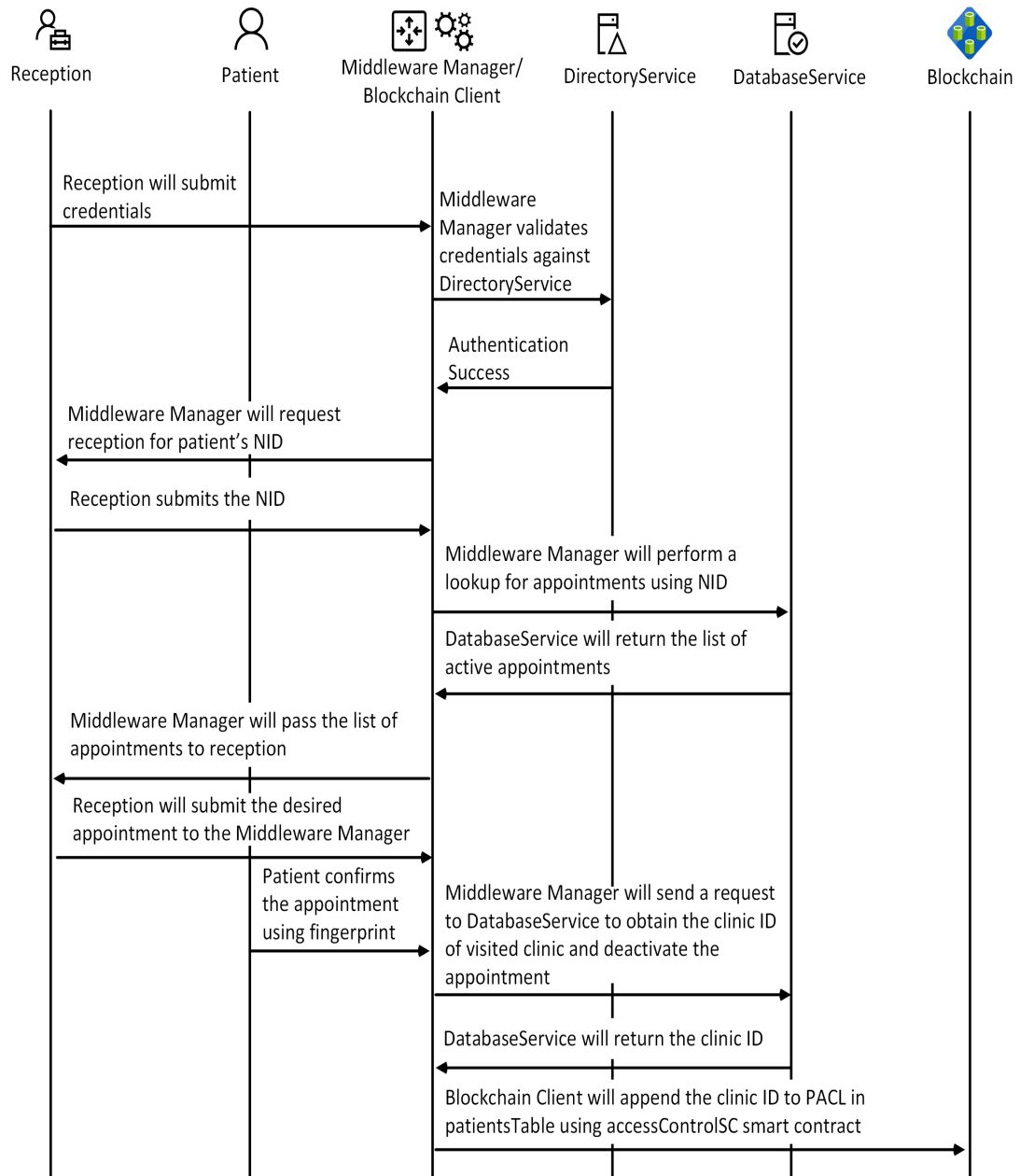


Figure 13: Booking Confirmation and Access Granting Summary

#### 2.2.4 Doctor's EHR Access

In the SDHCARE system, doctors can have read and/or write access to EHRs, as detailed in Section 2.1.3.6. This access can be granted by patients using their fingerprints, as described in Section 2.2.3. EHRs are divided into EHRs' metadata, which is stored in `ehrHashTable` and indexed using patients' fingerprints hashes to

ensure unique identification, and EHRs' raw data, which is stored in the cloud store and indexed using the Merkle root hash of the EHRs. Using read access, doctors can obtain a list of accessible patients' EHRs from `ehrHashTable` and download the desired one(s) from the cloud store. Write access allows doctors to create new EHR metadata in `ehrHashTable` and upload the EHR raw data to the cloud store. The following summarizes doctors' EHR read process, which is also outlined in Figure 14.

1. The doctor uses Doctor UI to authenticate against the provider's `DirectoryService`.
2. After successful authentication, Doctor UI is redirected to Patient UI for the patient to submit their fingerprint.
3. Upon the patient's submission of their fingerprint, the Middleware Manager queries the `DatabaseService` for the doctor's clinic ID using the doctor's username.
4. Next, the Middleware Manager makes a call to `accessControlSC` through the Blockchain Client. This call uses the patient's fingerprint hash and the doctor's clinic ID to validate whether the doctor has read access to the patient's EHR.
5. If the clinic ID is listed in the PACL of the patient (by reading `patientsTable` using the fingerprint hash), `accessControlSC` grants the doctor read access.
6. Based on the authorized read request, the Blockchain Client calls `patientsSC` to find the list of EHRs stored in `ehrHashTable`. This list is returned to the Middleware Manager which passes the details to Doctor UI. The lookup is executed by `patientsSC` using the patient's fingerprint hash passed from the Blockchain Client.

7. Upon the doctor's selection of the desired record, the Middleware Manager initiates a call to the cloud store, through Cloud API Interface, to retrieve the EHR data (using the Merkle root hash associated with the selected record in the list).
8. Once the EHR data is received by the Middleware Manager, it performs an integrity check by comparing the stored Merkle hash of the selected record against the calculated Merkle hash of the received data.
9. If the hash is valid, the Middleware Manager passes the data back to the doctor. Otherwise, an integrity failure alert is triggered.

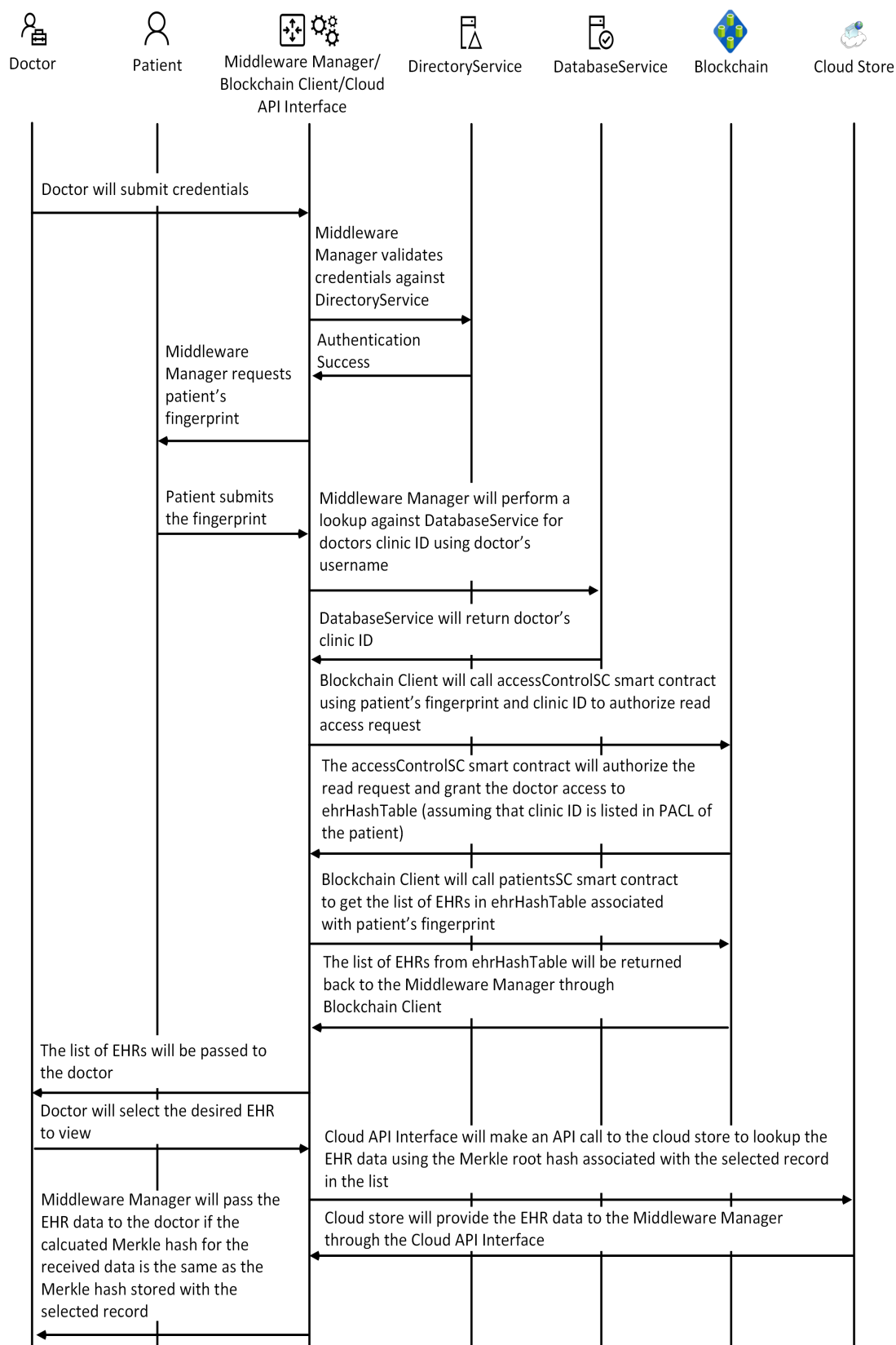


Figure 14: Summary of the EHR Read Process

Similar to the read process for EHRs, the process by which doctors write EHRs by doctors is outlined in Figure 15 and summarized as follows:

1. The doctor uses Doctor UI to authenticate against the provider's DirectoryService.
2. After successful authentication, the doctor adds details and/or attach files to the EHR.
3. The patient verifies the details of the EHR and confirms acceptance by submitting their fingerprint.
4. Once submitted, the Middleware Manager looks for the doctor's clinic ID in DatabaseService using the doctor's login.
5. Once the clinic ID is returned from DatabaseService, the Middleware Manager uses the Blockchain Client to call accessControlSC requesting authorization to write EHR metadata to ehrHashTable. The authorization request submits the patient's fingerprint hash and clinic ID to accessControlSC to verify if the clinic ID is listed in the PACL of the patient.
6. Upon successful authorization, the Middleware Manager calculates the Merkle root hash of the EHR.
7. Next, the Blockchain Client call patientsSC to write EHR metadata into ehrHashTable, which includes EHR details and the Merkle root hash. This is indexed using the patient's fingerprint hash.
8. Finally, the Middleware Manager uses the Cloud API Interface to upload the EHR data to the cloud store with the name of the EHR as the Merkle root hash.



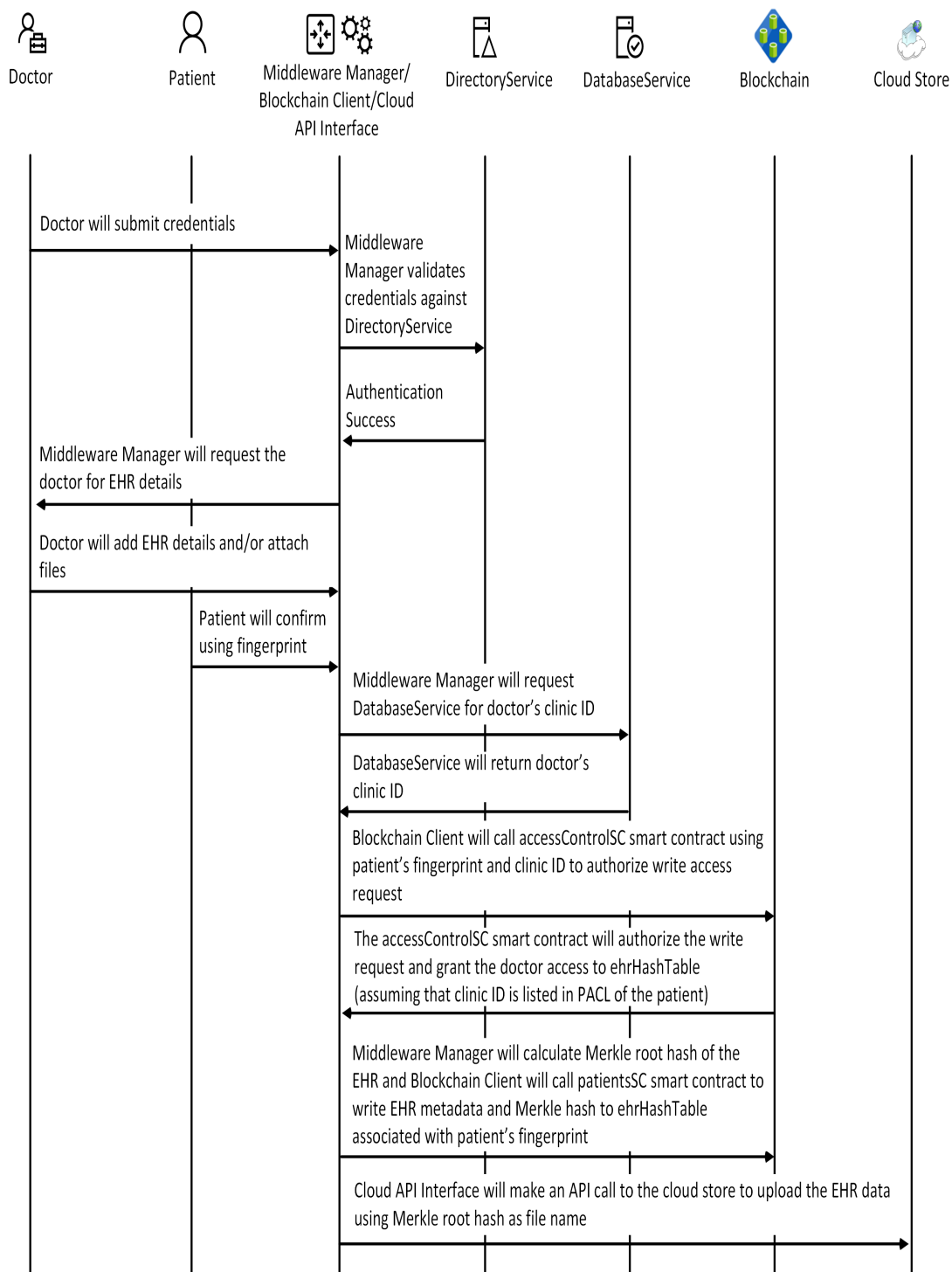


Figure 15: Summary of the EHR Write Process

## Chapter 3: Implementation

This chapter discusses the implementation of the SDHCARE prototype according to the proposed design functions. The chapter covers the selection process for the technologies used in the prototype and provides an overview of them. The final section of this chapter outlines the high-level implementation of the SDHCARE prototype. The low-level implementation and coding of the SDHCARE prototype are described in detail in the appendices.

### 3.1 Prototype Components

This section provides an overview of the components used to build our prototype. These components were selected based on the following criteria:

- Simplicity of the component setup and configuration to implement the required functions in SDHCARE.
- Interoperability capabilities of the component with other technologies to simulate the overall design of SDHCARE.
- Feature richness and built-in security capabilities of the component that are in line with SDHCARE requirements.
- Stability, reliability, and operational consistency of the component.

According to the proposed SDHCARE design, which was described in Chapter 2, the following technologies were selected for the prototype:

- Django 3.0 Web Development Platform: This was used to build the required web interfaces for Doctor UI, Admin UI, and Reception UI. Django is a Python-based platform that has built-in directory services.

This feature allows Django to integrate seamlessly with customized Python modules to provide additional functions such as API integration with Ethereum.

- Python 3.8: The Python interpreter is at the core of the SDHCARE prototype and provides the following functions:
  - Django web development coding language.
  - Interaction with Django built-in directory services.
  - API integration with the Blockchain and cloud store.
  - SQL interface with healthcare provider database.
  - Integrity validation and hashing of EHRs.
- SQLite 3: SQLite was used to implement the DatabaseService module local to the healthcare provider, mainly for storing information, including departments' IDs, appointments, and a provider's Blockchain details. SQLite has a native database connector with Django through Python.
- Ethereum Blockchain: This is the public Blockchain technology used to provide immutable smart contracts and immutable tables. Ethereum was chosen to extend the accessibility to EHRs at large scale, including cases in which patients relocate to different geographic areas. In such cases, the new geo-healthcare provider can connect to Ethereum and request access to a patient's EHR.
- Microsoft (MS) Azure Files: This service is hosted on the public cloud to store EHRs. The selection of MS Azure Files was due to its simple accessibility and usability in addition to its independence from the format of EHRs. MS Azure Files uses an SMB protocol to provide

secure communication between on-premise infrastructure and the cloud [24].

### 3.2 Django Architecture

This section covers the basics of Django web development architecture to enable an understanding of the setup and configuration of the SDHCARE prototype. Django consists of frontend and backend layers. The frontend layer consists of HTML templates with which clients interact. The backend layer, on the other hand, gets inputs from templates and performs the programmed function accordingly. Figure 16 shows a block diagram of Django architecture.

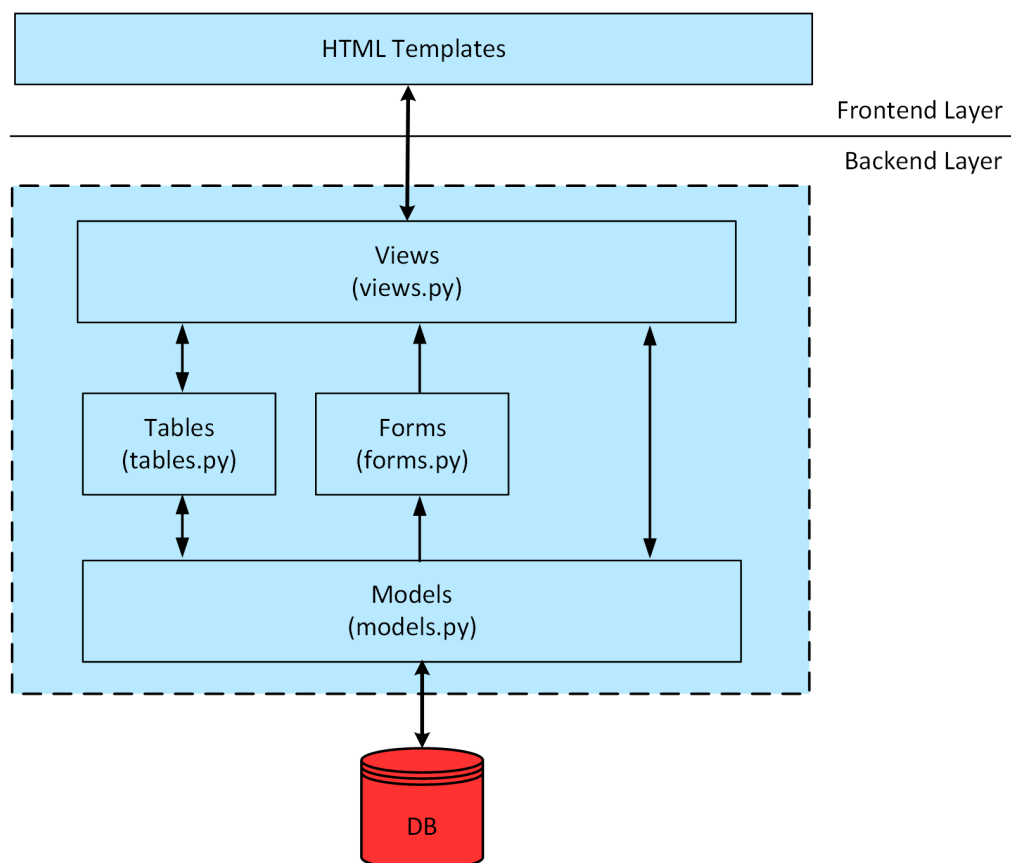


Figure 16: Django Web Development Architecture

Django simplifies HTML coding using Forms, which are Python functions defined in `forms.py` file. These functions specify HTML input fields and their labels, types, and maximum lengths, etc. (Forms do not include HTML styles and javascripts). The actual HTML pages presented to clients are combined versions of style sheets, javascripts, bootstraps, and other elements defined in HTML template files and with inputs returned from `forms.py`.

The input values returned from the clients can be written into the database through Models (`models.py`). Models act as an abstraction layer, provided by Django and programmed using Python. They can translate Python instructions into database queries depending on the type of integrated database. Models can receive input values from HTML templates through Views (`views.py`). Additionally, Models can poll data from the database and pre-fill Forms inputs to be presented to clients through HTML templates (such as by dropdown selection).

Tables (`tables.py`) in Django are used to populate information from the database in table format and present them to clients. This simplifies the process of creating HTML tables compared to traditional HTML methods. Tables do not handle styles, as this is controlled through HTML templates.

Django uses Views to link Forms, Tables, and Models with Templates. Views control the logic of the Django web flow and how requests/responses are handled between clients and the web application. It is at this point that SDHCARE core functions are implemented. Additionally, Views allow the import of custom Python models for extended functionalities that are not present in Django.

### 3.3 System Implementation

This section outlines the implementation steps of the SDHCARE prototype and covers high-level implementation. The source code for Django, Python modules, and Ethereum smart contracts is included in the appendices.

#### 3.3.1 Building the Runtime Environment

The first step in building the SDHCARE prototype was setting up the runtime environment that hosts the prototype components (described in Section 3.1). These components are independent of operating-system. MacOS Catalina was selected as the OS hosting SDHCARE system in the healthcare provider to run Django, Python, and SQLite, and communicate with Ethereum and MS Azure. Figure 17 shows the used version of MacOS.



Figure 17: MacOS Software Version for SDHCARE

The next step was to download and install Python 3.8 and PyCharm 2019.3.2 IDE. From PyCharm IDE, a new virtual environment was created to use Python 3.8 as an interpreter for Django and custom Python modules. This was followed by creating a new project in PyCharm called SDHCARE to run on the created virtual environment, i.e., interpreted using Python 3.8 as shown in Figure 18.

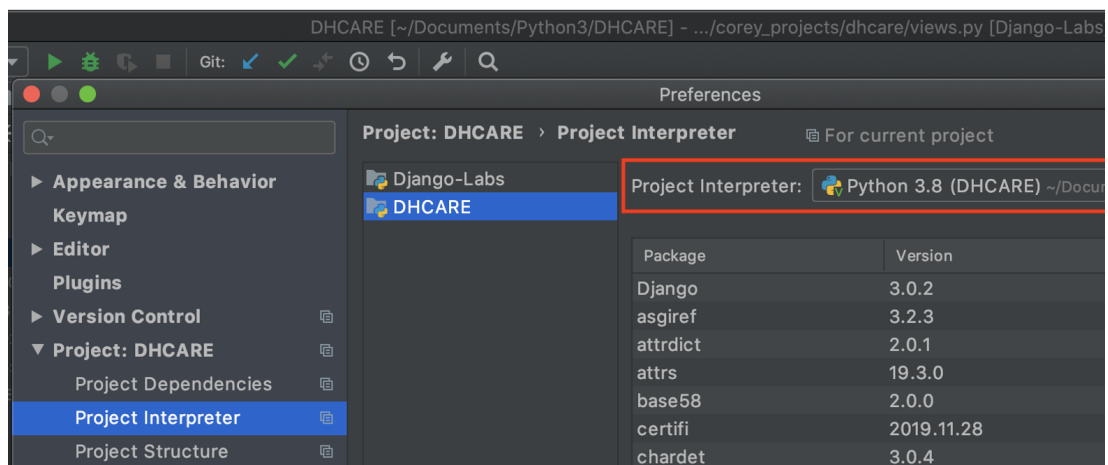


Figure 18: PyCharm SDHCARE Project with Python 3.8 Interpreter

Following the creation of the SDHCARE project in PyCharm, a list of required Python libraries was installed. Table 5 summarizes this list of libraries. Python uses the PIP3 utility to install external libraries from the internet.

Table 5: Python Libraries Required for SDHCARE

Library Name	Purpose
django	Web development framework; this installation includes SQLite 3
django-Tables2	For formatting and styling tables in Django
crispy	For formatting and styling HTML templates in Django
web3	For API communication with Ethereum
azure-storage-file	For API communication with MS Azure

### 3.3.2 Initializing SDHCARE Web and Database Components

After preparing the runtime environment for hosting SDHCARE, the next step was initializing the Django web framework and database. The initialization was done in the following order to ensure the successful running of Django:

1. Initialize SQLite3 DB to be ready for storing the provider's Ethereum information and clinics' information. This is done by running the following commands from the PyCharm SDHCARE project terminal:  

```
python manage.py makemigrations
```

```
python manage.py migrate
```
2. Create a Django admin user to administrate the Django management console, including account creation in the Django built-in directory service. These accounts represent doctors, nurses, officers, and receptionists. Additionally, the admin user populates the SQLite3 database with information about the healthcare provider's clinics and Ethereum information. The command for creating an admin user is:



python manage.py createsuperuser

3. Create SDHCARE user accounts and groups by navigating to <http://localhost:8000/admin>, signing in using the admin account, adding groups for different permissions, adding new users, and assigning users to their respective groups. Figure 19 shows a sample of users, groups, and their assignments.

Home > Authentication and Authorization > Users

Select user to change

Action:   0 of 6 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS
<input type="checkbox"/>	admin	admin@dhcare.com
<input type="checkbox"/>	pA-doctor-dental	
<input type="checkbox"/>	pA-doctor-ortho	
<input type="checkbox"/>	pA-doctor-pediatrics	
<input type="checkbox"/>	pA-officer	
<input type="checkbox"/>	pA-reception	

6 users

Home > Authentication and Authorization > Groups

Select group to change

Action:   0 of 3 selected

<input type="checkbox"/>	GROUP
<input type="checkbox"/>	DHCARE-Admins
<input type="checkbox"/>	DHCARE-Doctors
<input type="checkbox"/>	DHCARE-Reception

3 groups

Permissions

☒ Active  
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

☐ Staff status  
Designates whether the user can log into this admin site.

☐ Superuser status  
Designates that this user has all permissions without explicitly assigning them.

Groups:

Available groups ?

Filter

DHCARE-Admins  
DHCARE-Reception

Chosen groups ?

DHCARE-Doctors

Figure 19: Example of Users, Groups, and Group Assignment

4. Create local database tables for the SDHCARE healthcare provider. These tables are stored in the SQLite3 database with the following structures:

- Appointments Table: Name Column (patient name), NID, Date Column, Time Column, and Department Column (clinic to be visited).
- Departments Table: Code Column (clinic code) and Name Column (clinic name).
- Provider Table: OHP Column (Ethereum public key) and Secret Column (Ethereum private key).

The Python code for creating these tables was written in models.py as shown below.

```
class department(models.Model):
    code = models.IntegerField(unique=True, primary_key=True)
    name = models.CharField(max_length=100)

class provider(models.Model):
    ohp = models.CharField(max_length=100, primary_key=True)
    secret = models.CharField(max_length=100)

class appointment(models.Model):
    name = models.CharField(max_length=100)
    nid = models.IntegerField()
    date = models.DateField(default=timezone.now)
    time = models.TimeField(default=timezone.now)
    department_code = models.ForeignKey('department',
on_delete=models.CASCADE)
```

5. From the Django admin console, populate departments and provider database tables with provider's clinics' details and Ethereum details, respectively (appointments database is populated by the receptionist, as described later). Figure 20 shows the created tables from the Django admin console.

The screenshot displays the SDHCARE web application interface. On the left, a sidebar contains navigation links: **DHCARE**, **Appointments**, **Departments** (highlighted), and **Providers**. Below the sidebar, a breadcrumb trail reads **Home > Dhcare > Departments > Dental**. The main content area is titled **Change department** and includes two input fields: **Code:** with the value **103**, and **Name:** with the value **Dental**. A red **Delete** button is located below these fields. On the right, a section titled **Select department to change** features an **Action:** dropdown menu and a **Go** button. Below this is a list of departments, each with a checkbox and a label: **DEPARTMENT**, **Dental**, **Orthopedics**, **Hematology**, **Pediatrics**, **Officer**, **Reception**, and **Admin**. At the bottom of this list, it states **7 departments**.

Figure 20: Example of SDHCARE Databases

### 3.3.3 Building SDHCARE Django Code

After the successful initialization of Django, the next step was writing the Django Python code to perform the required functions of the SDHCARE prototype. As mentioned at the beginning of this chapter, this section covers only the high-level coding structure. The low-level coding is available in the appendices.

The first step in Django coding was developing the HTML templates for the frontend layer. The approach for HTML coding was based on developing a base template (base.html) containing all shared components across all pages, such as header, footer, title, and styles. Any child HTML template has page-specific components combined with the base template, presented to the user. Figure 21 summarizes the HTML coding approach along with all the HTML templates.

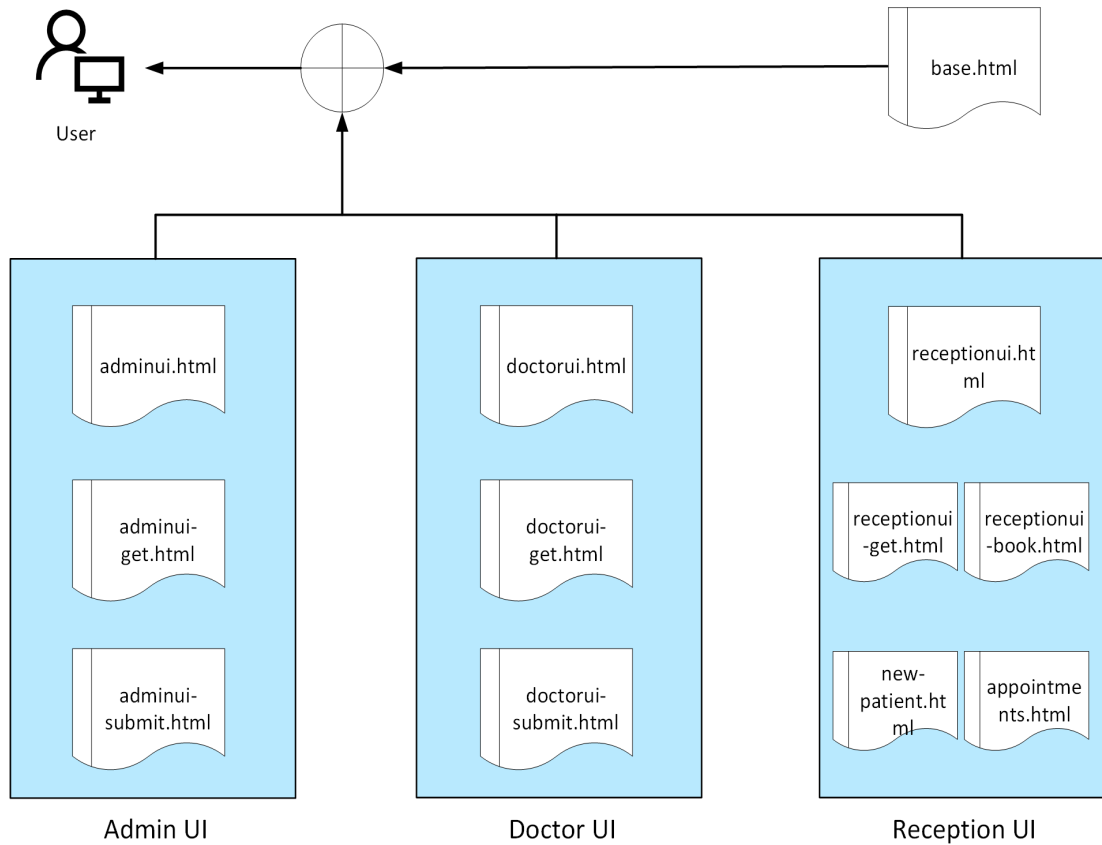


Figure 21: Summary of HTML SDHCARE Template Structure

As summarized in Figure 21, each UI has a set of specific HTML templates that are combined with `base.html` before being presented to the user.

After creating the HTML templates, the next step was building the functions in `forms.py` (for the input fields to be presented with each template) and associating each form's function with its corresponding HTML template through functions in `views.py`. Figures 22, 23, and 24 provide summaries of `forms.py`, `views.py`, and the corresponding UI.

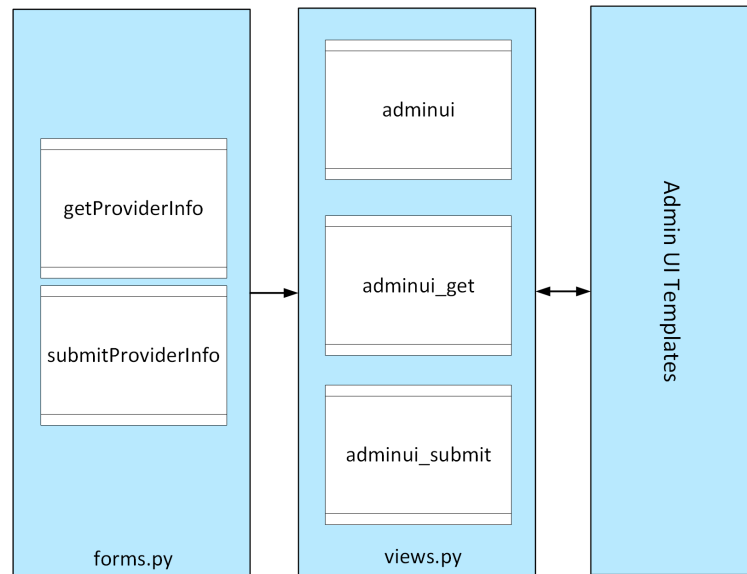


Figure 22: Admin UI with Forms and Views

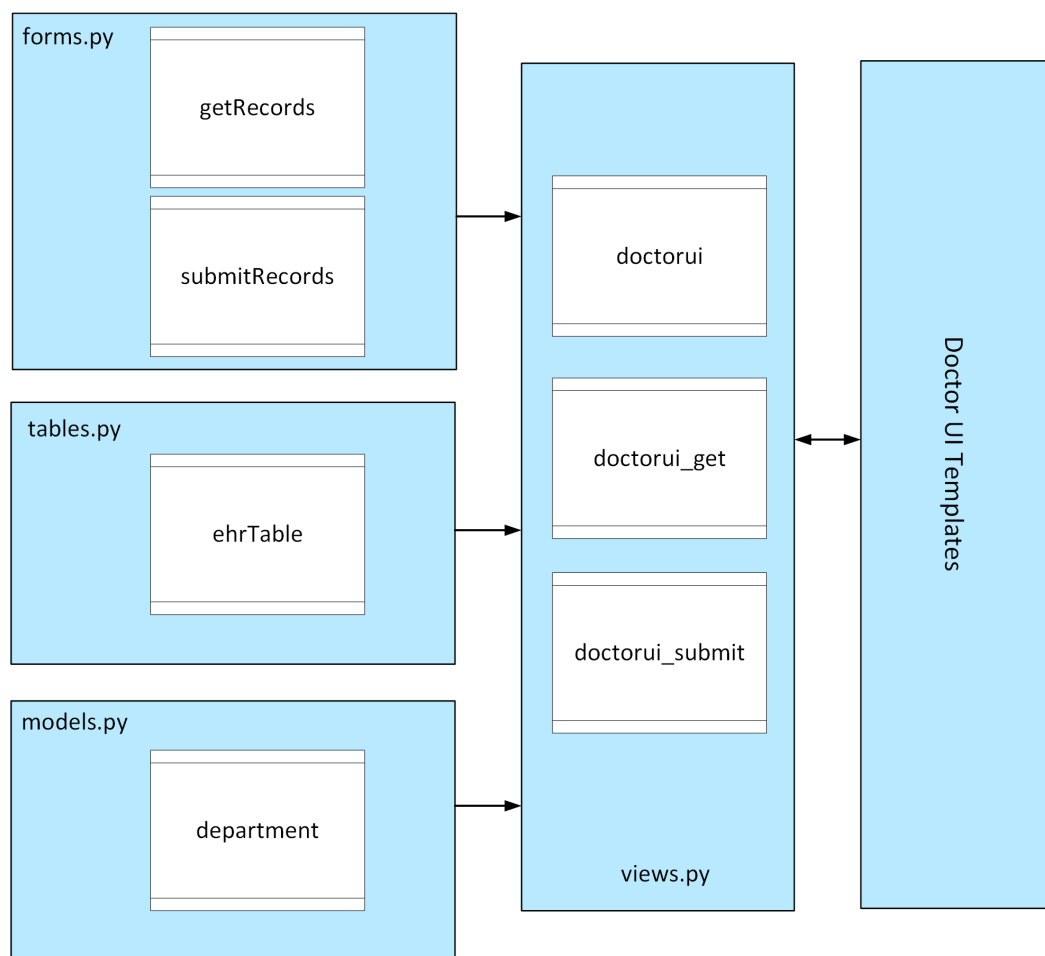


Figure 23: Doctor UI with Forms and View

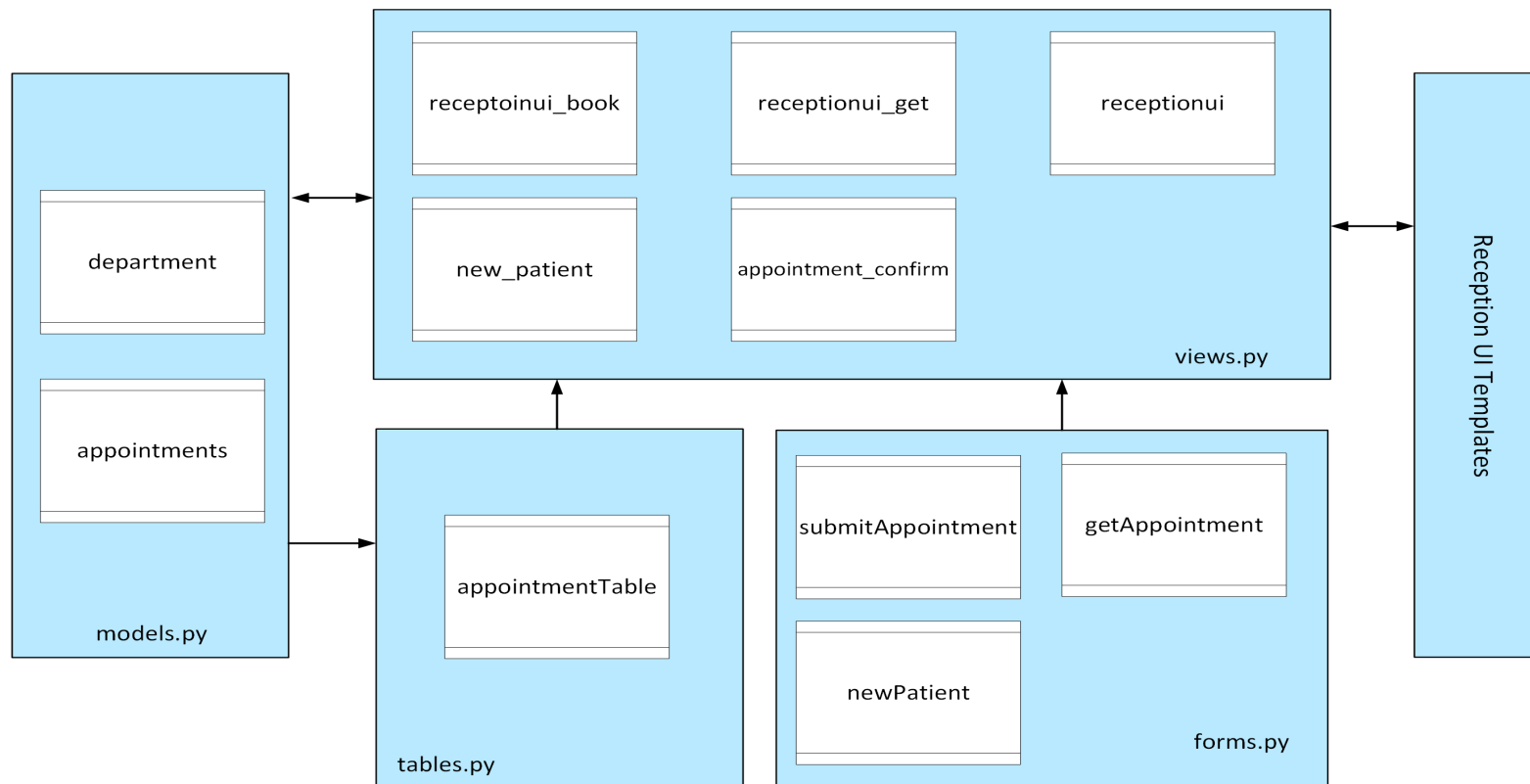


Figure 24: Reception UI with Forms and Views

Below is a sample of code in views.py, which links adminui-get.html with the getProviderInfo function from forms.py. The page should display one field to the user to enter ‘Hospital Ethereum Address’ as shown in Figure 25.

```
views.py
*****

def adminui_get(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
        request:
        form = getProviderInfo(request.POST)
        # check whether it is valid:
        if form.is_valid():
            account_address = form.cleaned_data.get('OHP_Eth')
            provider = dnsSC_get(account_address)
            context = {
                'provider': provider
            }

            return render(request, 'SDHCARE/adminui-get.html', context)

    # if a GET (or any other method) we'll create a blank form
    else:
        form = getProviderInfo()

forms.py
*****

class getProviderInfo(forms.Form):
    OHP_Eth = forms.CharField(label='Hospital Ethereum Address',
max_length=100)
```

DHCARE Home About

### Hospital Information

Hospital Ethereum Address\*

This field is required.

Get Info

Figure 25: Healthcare Provider’s getProviderInfo Page

Access to each set of UIs is controlled using role-based access control (RBAC) implemented by Python decorators. The decorator obtains the session username, verifies its group membership (which was configured during Django initialization), and allows access only if the user is a member of the required group. Below is a sample code of using decorators to limit doctors' access to Doctors UI only. These decorators are implemented in views.py.

```
@custom_user_passes_test(lambda u: Group.objects.get(name='SDHCARE-Admins')
in u.groups.all())
def adminui(request):

@custom_user_passes_test(lambda u: Group.objects.get(name='SDHCARE-Admins')
in u.groups.all())
def adminui_get(request):

@custom_user_passes_test(lambda u: Group.objects.get(name='SDHCARE-Admins')
in u.groups.all())
def adminui_submit(request):
```

Another access control mechanism is applied to grant permissions to access patients' EHRs, using decorators and smart contracts. Clinics are granted permissions to access patients' EHRs only if the session user is a member of the SDHCARE-Reception group and the patient submits a valid fingerprint. A third form of access control is applied for doctors' access to EHRs and is covered in Section 3.3.4.

The last step in Django coding was linking models to UIs through views. In Figures 22, 23, and 24 the set of functions in models.py are mapped to their respective UIs to have information read from the database and displayed to clients, such as a patient's appointments or a provider's OHP address, or to write information such as booking a new appointment into the database. Some database information, such as appointments, is formatted in tables, hence the information from models.py is passed to views.py through tables.py. Below is a sample code for formatting appointment data.



```

tables.py
*****

class appointmentTable(tables.Table):
    department_code = tables.Column(verbose_name='Clinic')
    #nid = tables.Column(verbose_name='National ID')
    # id = tables.Column(visible=False)
    id = tables.CheckBoxColumn(accessor='id')

    class Meta:
        model = appointment
        template_name = "django_tables2/bootstrap4.html"

models.py
*****

class appointment(models.Model):
    name = models.CharField(max_length=100)
    nid = models.IntegerField()
    date = models.DateField(default=timezone.now)
    time = models.TimeField(default=timezone.now)
    department_code = models.ForeignKey('department',
on_delete=models.CASCADE)

    def __str__(self):
        return self.name

views.py
*****

def receptionui_get(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
        request:
        form = getAppointments(request.POST)
        # check whether it is valid:
        if form.is_valid():
            nid = form.cleaned_data.get('nid')
            table = appointmentTable(appointment.objects.filter(nid=nid))
            return render(request, 'SDHCARE/receptionui-get.html',
{"table": table})
        else:
            form = getAppointments()

    return render(request, 'SDHCARE/receptionui-get.html', {'form': form})

```

The following set of tables (Tables 6, 7, and 8) summarize the functions configured in Views, Forms, and Tables. These functions describe how SDHCARE design requirements are implemented in Django. The functions in Models were described in the previous section that outlines Django SQLite3 DB initialization.

Table 6: Summary of Views Functions

Name	Description
home	To render home page template to users
about	To render about page template to users
adminui	<ul style="list-style-type: none"> <li>Verifies that the session user is a member of the SDHCARE-Admins group</li> <li>Renders the base and adminui templates for admins to select 'Enter Hospital Information' or 'Get Hospital Information'</li> </ul>
adminui_get	<ul style="list-style-type: none"> <li>Verifies that a session user is a member of the SDHCARE-Admins group</li> <li>Reads admin OHP input and obtains the hospital information stored in Ethereum providersTable for that OHP; this information is returned to the admin</li> </ul>
adminui_submit	<ul style="list-style-type: none"> <li>Verifies that the session user is a member of the SDHCARE-Admins group</li> <li>Reads admin input (hospital name and web address) and stores the information in Ethereum providersTable using OHP as the indexing key; OHP is obtained from the address of the transaction sender</li> </ul>
receptionui	<ul style="list-style-type: none"> <li>Verifies that the session user is a member of the SDHCARE-Reception group</li> <li>Renders the base and receptionui templates for reception to select 'New Patient', 'Get Patient Appointments', or 'Book New Appointment'</li> </ul>
receptionui_get	<ul style="list-style-type: none"> <li>Verifies that the session user is a member of the SDHCARE-Reception group</li> <li>Accepts a patient's fingerprints and uses it as an index key to obtain all active appointments associated with that patient</li> </ul>
appointment_confirm	Performs two-factor validation by verifying a patient's fingerprint and reception group membership; if both are valid, the clinic is granted access to the patient's EHRs; this access is stored in the Ethereum patientsTable and covers all doctors in that clinic

Table 6: Summary of Views Functions (Continued)

Name	Description
receptionui_book	<ul style="list-style-type: none"> <li>• Verifies that the session user is a member of the SDHCARE-Reception group</li> <li>• Books new appointments for patients and stores them in the appointments database, which is indexed using the patient's NID</li> </ul>
new_patient	<ul style="list-style-type: none"> <li>• Verifies that the session user is a member of the SDHCARE-Reception group</li> <li>• Creates a new patient record in the Ethereum patientsTable</li> </ul>
doctorui	<ul style="list-style-type: none"> <li>• Verifies that the session user is a member of the SDHCARE-Doctor group</li> <li>• Renders the base and doctorui templates for the doctor to select 'Get Patient EHRs' or 'Submit Patient EHRs'</li> </ul>
doctorui_get	<ul style="list-style-type: none"> <li>• Verifies that the session user is a member of the SDHCARE-Doctor group</li> <li>• Verifies that the doctor's clinic is granted access to the patient's EHRs (using accessControlSC, which is covered in Section 3.3.4)</li> <li>• Displays a list of EHRs and hashes to the doctor, which are indexed using patients' fingerprints (from the Ethereum ehrHashTable); these records are formatted in a table before being passed to doctors</li> <li>• Once the doctor selects an EHR, this verifies that the cloud EHR's hash is the same as the hash in ehrHashTable</li> <li>• Retrieves the EHR from Cloudstore</li> </ul>
doctorui_submit	<ul style="list-style-type: none"> <li>• Verifies that the session user is a member of the SDHCARE-Doctor group</li> <li>• Verifies that the doctor's clinic is granted access to a patient's EHRs using the patient's fingerprint (using accessControlSC, which is covered in Section 3.3.4)</li> <li>• Creates an EHR based on information submitted by a doctor; the EHR is stored in the Ethereum ehrHashTable. A Merkle root hash is calculated for the EHR and uploaded with the record</li> </ul>

Table 7: Summary of Forms Functions

<b>Name</b>	<b>Description</b>
getProviderInfo	Presents one input field to the provider's admin: the provider's Ethereum address (OHP)
submitProviderInfo	Presents two input fields to the provider's admin: provider name and provider web address
bookAppointment	Presents to the receptionist the following fields: name, fingerprint, date, time, dropdown for the clinics
getAppointments	Present to the receptionist a single field: the patient's fingerprint
newPatient	Presents to the receptionist the following fields: name, date of birth, and fingerprint
submitRecords	Presents to the doctor the following fields: patient name, patient fingerprint, record name, record date, and record description
getRecords	Presents to the doctor one field: fingerprint

Table 8: Summary of Tables Functions

<b>Name</b>	<b>Description</b>
appointmentTable	Formats the appointments retrieved from the local database in a table before posting them to the receptionist
ehrTable	Format the EHR list retrieved from the Ethereum ehrHashTable in a table before posting them to the doctor

Figures 26 to 32 are examples of the UIs displayed to different users. These pages represent the combined version of base.html and functional templates with some images rendered from static fills.

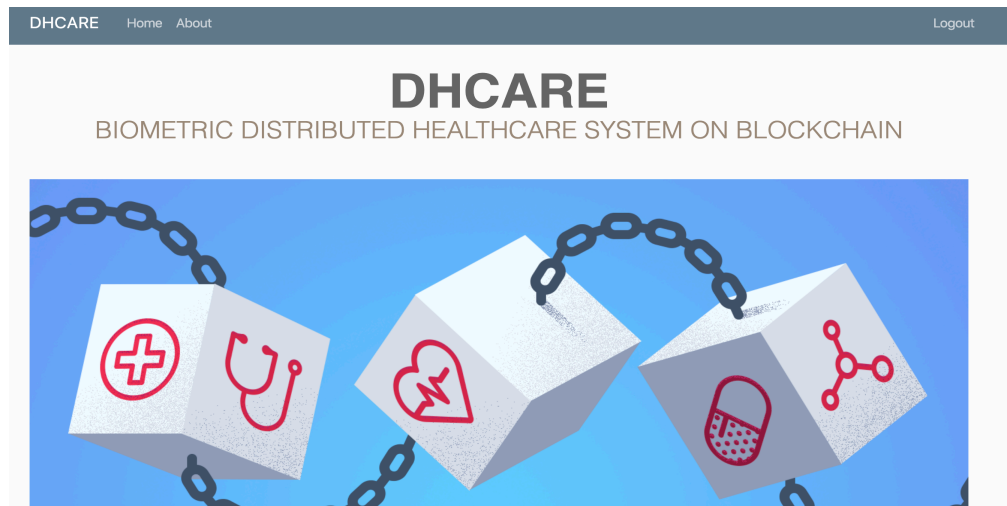


Figure 26: SDHCARE Home Page

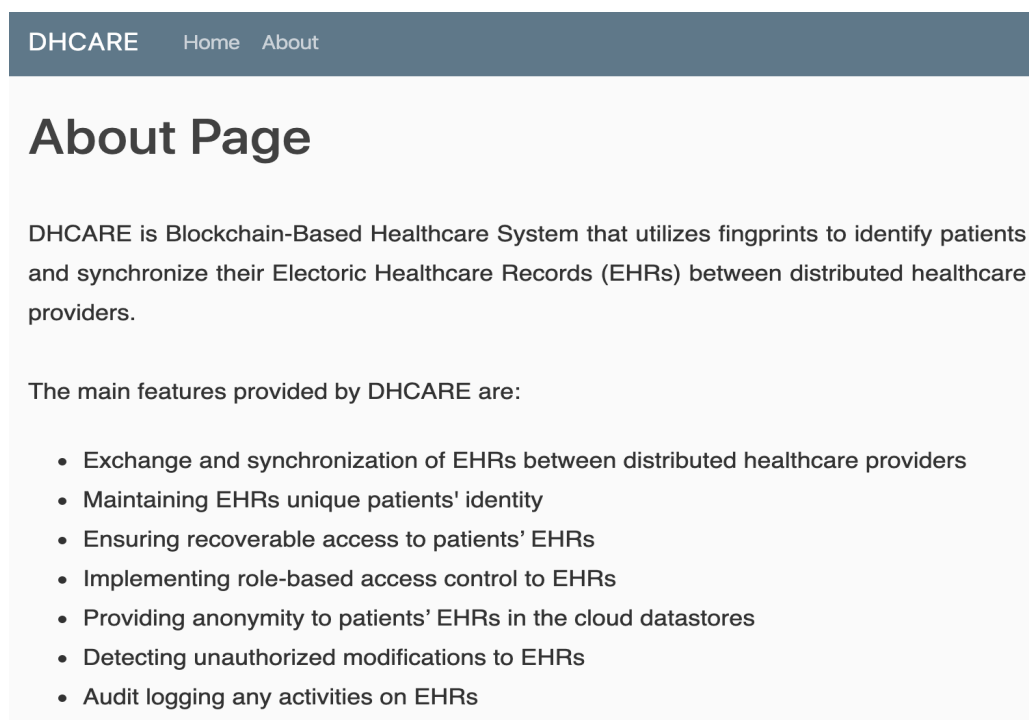


Figure 27: SDHCARE About Page

DHCARE

[Home](#)

[About](#)

Log In

Username\*

admin

Password\*

.....

Login

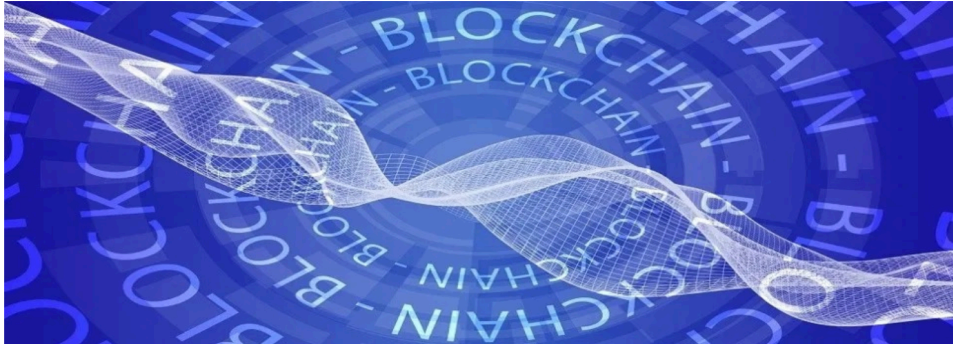
Figure 28: SDHCARE Login Page

DHCARE

[Home](#)

[About](#)

Admin Portal

A graphic featuring the word 'BLOCKCHAIN' repeated in a circular pattern on a blue background, with a white, wavy, wireframe-like structure overlaid.

Get Hospital Information

Submit Hospital Information

Figure 29: SDHCARE Admin UI Page

DHCARE Home About

### Hospital Information

Hospital Name\*

**This field is required.**

Hospital Web Address\*

**This field is required.**

[Submit Info](#)

Figure 30: SDHCARE Admin UI Submit Page

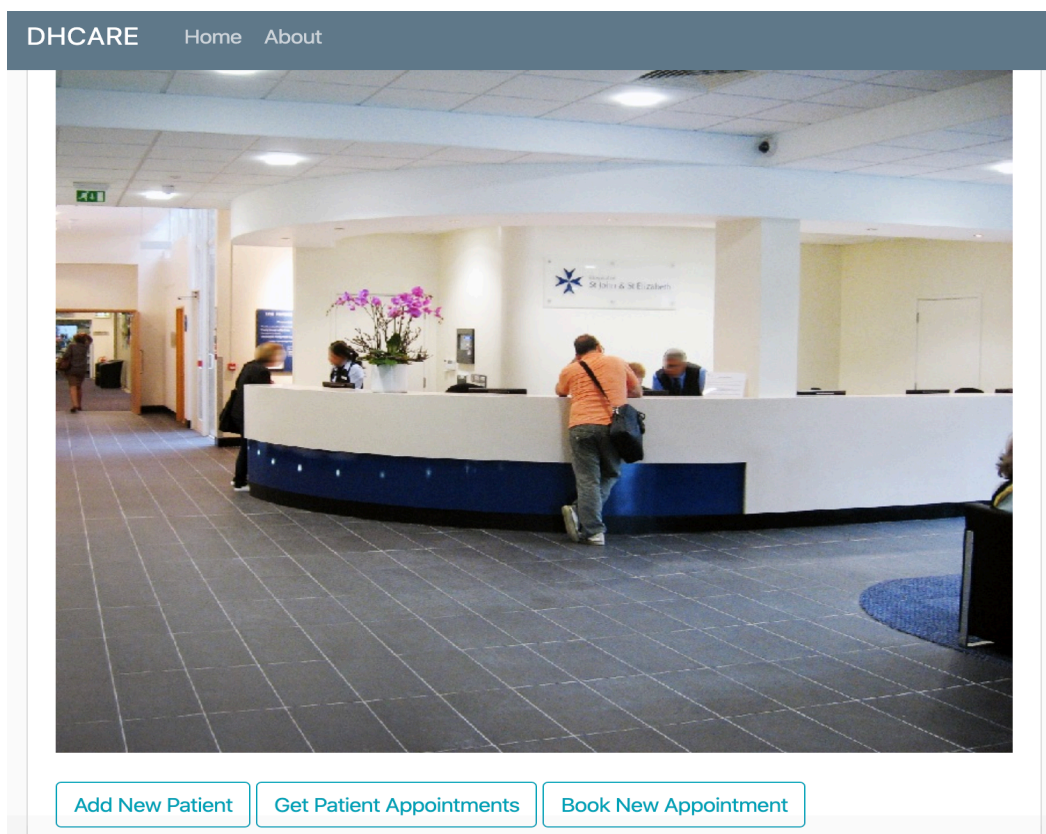


Figure 31: SDHCARE-Reception UI Page

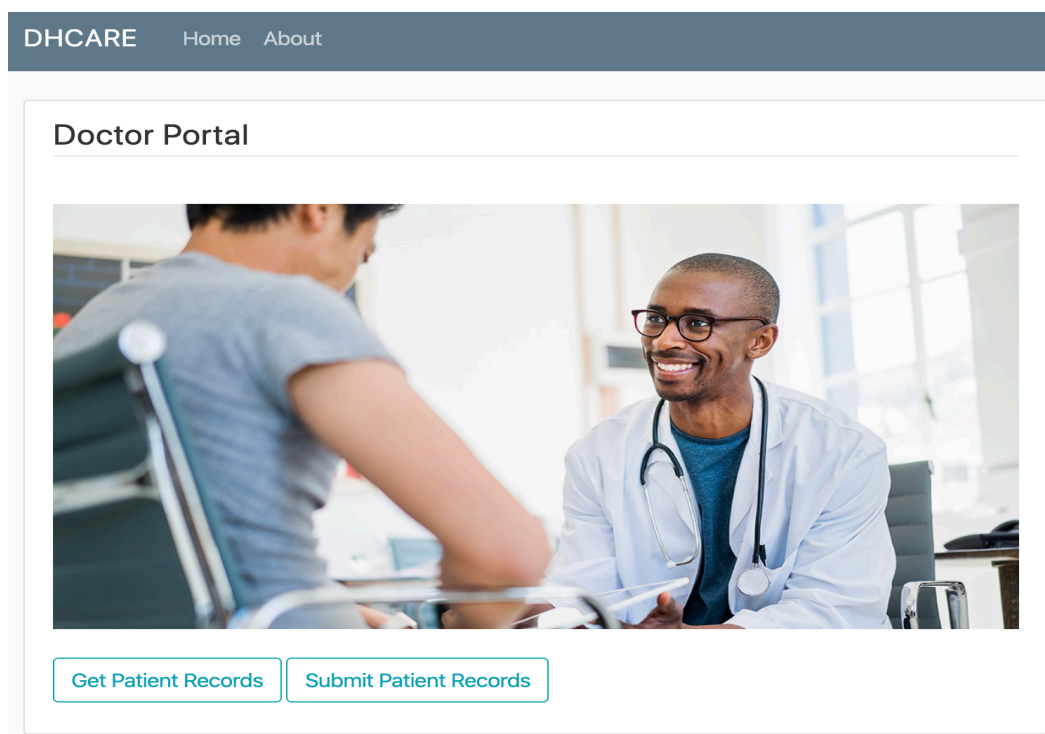


Figure 32: SDHCARE Doctor UI Page

### 3.3.4 Building Ethereum Smart Contracts

The prototype for DHCARE has three smart contracts, known as `dnsSC`, `patientsSC`, and `accessControlSC`, which provide the functions described in Section 2.1.3. The main reason for creating three smart contracts instead of a combined one is to provide flexibility in extending the functionality of the SDHCARE prototype through inheriting and importing smart contract functions.

The `dnsSC` has two functions:

- `createProvider`: This function takes two string inputs for the healthcare provider's name and web address. These values are stored in the `providersTable` immutable table, which is indexed by the provider's Ethereum address (OHP). In the case of an existing provider, the function returns an exception error.



- **getProvider:** This function takes an address input (OHP) and returns two string variables that represent the provider name and web address, which are stored in `providersTable`.

Similar to `dnsSC`, `accessControlSC` has two functions:

- **addClinic:** This function takes two string inputs representing the patient's fingerprint hash and the clinic ID. The clinic ID is polled from the SQLite3 department database. Both strings are stored in an array indexed by the patient's fingerprint hash that represents all the clinics that can access the patient's EHRs. This array is part of `patientsTable`.
- **grantClinicAccess:** This function takes two string inputs, namely the patient's fingerprint hash and the clinic ID. It performs a lookup in the `patientsTable`, using the patient's fingerprint hash array to determine whether the clinicID is listed. If the clinicID is listed in the array, it returns 'True', which allows the doctor to access the patient's EHR. Otherwise, it returns 'False', which denies doctor's access.

The `patientsSC` includes the following four functions:

- **createPatient:** This function accepts string inputs for the patient's fingerprint hash, provider Ethereum address, and reception ID. It stores this information in `patientsTable` indexed by fingerprint hash.
- **getPatient:** This function accepts a string input of the patient's fingerprint hash and returns the patient's stored values in `patientsTable` (i.e., provider's address and reception ID).

- **createEHR:** This function accepts the patient's fingerprint hash, EHR name, date, status, and Merkle root hash. It stores the values in `ehrHashTable` indexed by fingerprint hash.
- **getEHR:** This function accepts a fingerprint hash input string and returns the patient EHR list. For each EHR, the returned values are name, date, Merkle root hash, and EHR status.

The smart contracts were deployed in Ethereum using MetaMask soft wallet, and each contract is allocated a unique address for communication. Figure 33 is a summary of transactions between a healthcare provider and Ethereum, including smart contract deployments (The account of the test provider was used to deploy smart contracts for demo purposes. In a real-life implementation, such contracts should be deployed once by the owner of the project).

## Transactions

For [0x4dF59bA9e77816A8D73F40E617b0421Be333dA79](#)

Sponsored:  **Fortmatic** - Build Ethereum web3 dApps without browser extensions or mobile wallets. [Get Started](#) <sup>①</sup>

A total of 112 transactions found

[First](#) [<](#) [Page 2 of 3](#) [>](#) [Last](#) [⋮](#)

Txn Hash	Block	Age	From	To	Value	[Txn Fee]
<a href="#">0xe5953aa5a28068...</a>	<a href="#">7243628</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">Contract Creation</a>	0 Ether	0.00067965
<span>!</span> <a href="#">0x6744b868c2965a...</a>	<a href="#">7243576</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">Contract Creation</a>	50 wei	0.00016355
<a href="#">0xa5189525a315a3...</a>	<a href="#">7243506</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">0x4daeb7f9aa72c25...</a>	0 Ether	0.00002735
<a href="#">0x808ff01919576e6...</a>	<a href="#">7243503</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">0x4daeb7f9aa72c25...</a>	0 Ether	0.00006512
<a href="#">0xd289b15b8dcb3c...</a>	<a href="#">7243500</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">Contract Creation</a>	0 Ether	0.00038036
<a href="#">0xa905037c2fccd9d...</a>	<a href="#">7243493</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">0x6bffe5d79d80708...</a>	0 Ether	0.00002735
<a href="#">0x450b639c177ca0...</a>	<a href="#">7243491</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">0x6bffe5d79d80708...</a>	0 Ether	0.00006512
<a href="#">0xf15d63ada42e76...</a>	<a href="#">7243489</a>	17 days 6 hrs ago	<a href="#">0x4df59ba9e77816...</a>	<span>OUT</span> <a href="#">Contract Creation</a>	0 Ether	0.00038037

Figure 33: Summary of Ethereum Transactions

### 3.3.5 Integrating Django with Ethereum

The integration between the Django web application and Ethereum was implemented using Web3 customized Python modules on the Django side and the Infura mining pool on the Ethereum side. Figure 34 is a summary of the integration between Django and Ethereum.

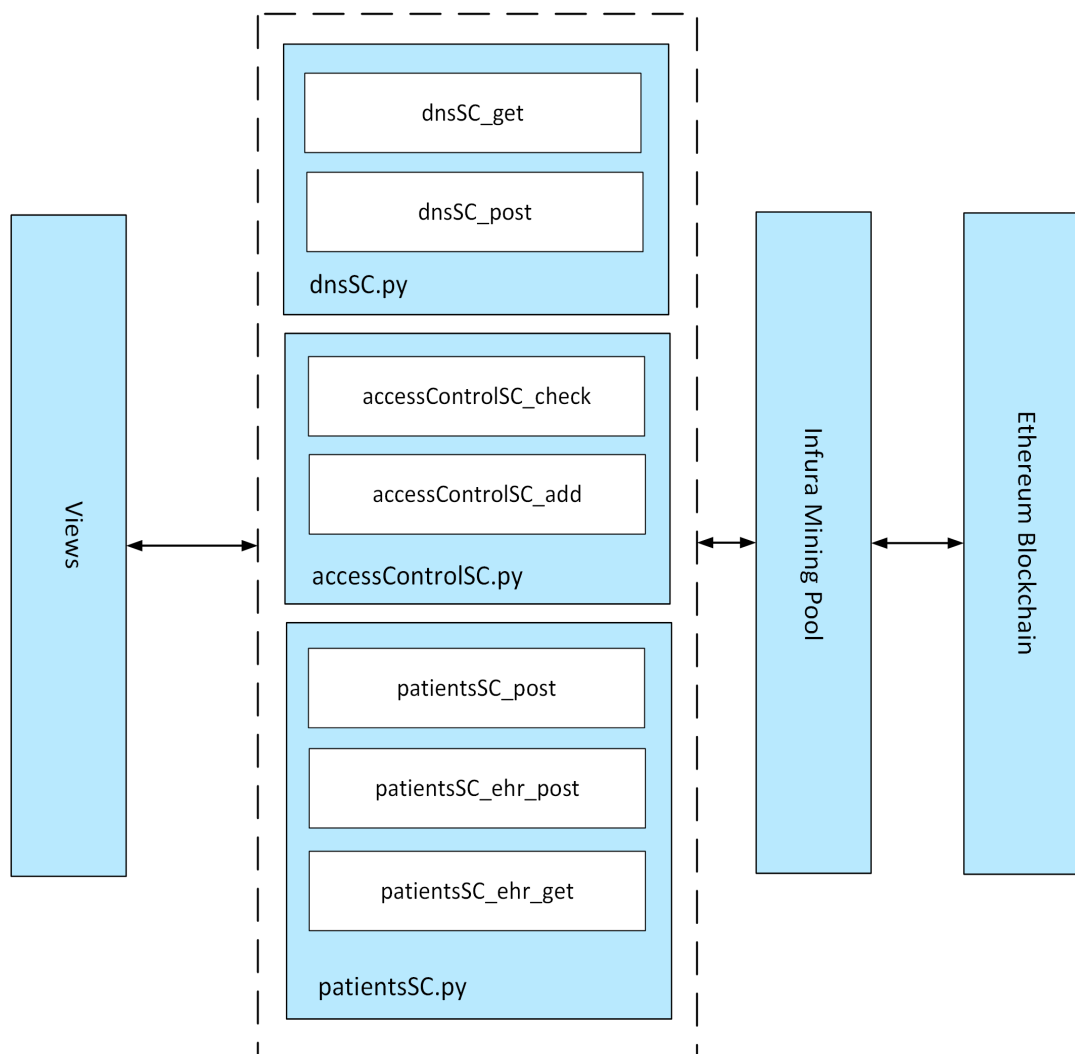





Figure 34: Integrating Django with Ethereum

To communicate with Ethereum Blockchain, the healthcare provider should contribute with a dedicated mining node running Ethereum mining software (e.g., Get

Ethereum, or GETH). A dedicated node is used to ensure patients' privacy. For the prototype, Infura mining pool was used, which offers mining nodes as a service to interact with Ethereum. The free version of Infura offers 100,000 Ethereum transactions within 24 hours. An Infura account was created, which provides a unique URL to communicate with Infura nodes for posting and reading blocks to and from Ethereum. Figure 35 shows the details of the Infura test account.

 DASHBOARD STATS

UPGRADE

EDIT PROJECT


NAME


Thesis


SAVE CHANGES

KEYS


PROJECT ID

db1690d7911842a6a0ec7690d08a0ca3

PROJECT SECRET 

515fd25627ee4a66be5d1fe05ac8f46d

ENDPOINT

ROPSTEN 


ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3

Figure 35: Infura Account Details

For Django Views to interact with Ethereum, custom Python modules were built utilizing Web3 APIs. Each Python module has functions to interact with the respective smart contract functions. Figure 36 summarizes the operation of the Python modules.

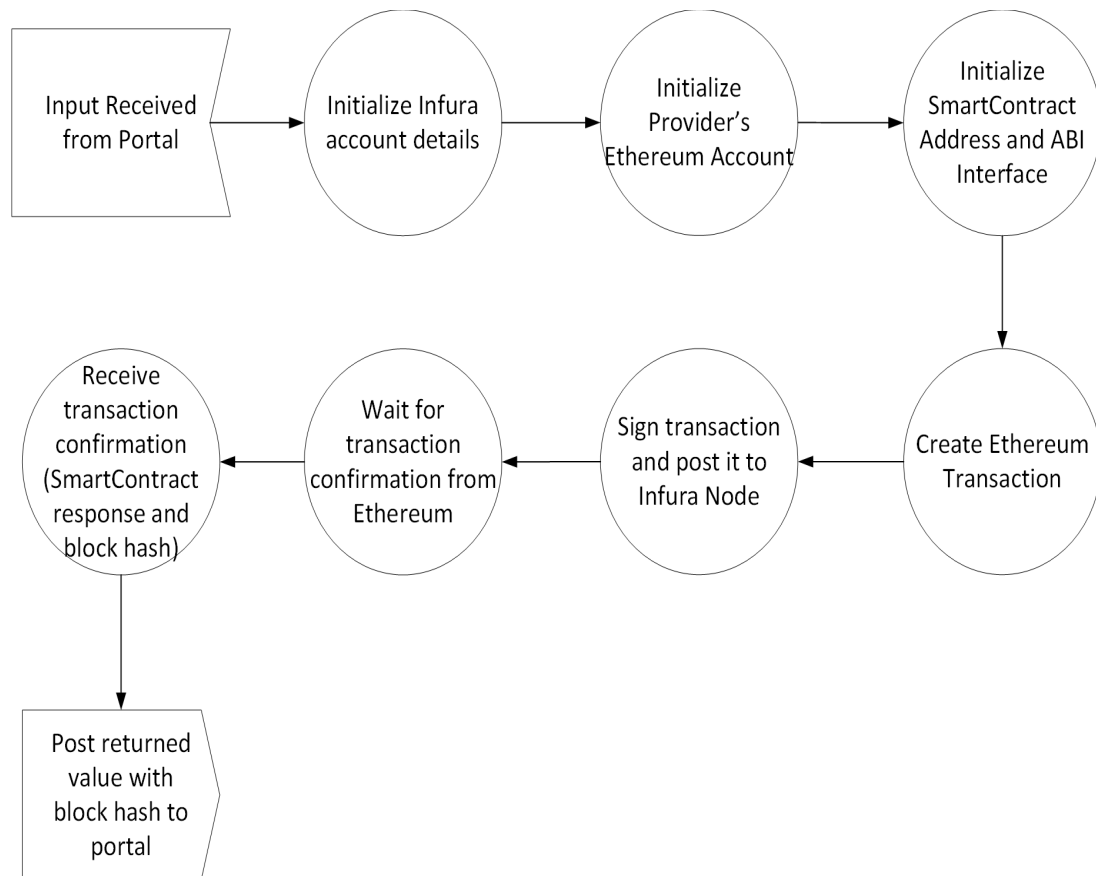


Figure 36: Custom Python Module Operation

## Chapter 4: Testing and Performance Evaluation

This chapter describes the testing and evaluation of SDHCARE prototype implementation from functional, security, and performance perspectives. Functional testing ensured that the prototype is operating as expected by design. Security testing validated the security measures implemented in the prototype to protect it against unauthorized access and modifications of EHRs. Performance evaluation measured the elapsed time required to gain read or write access to EHRs.

### 4.1 Functional Testing

The functional testing evaluated the SDHCARE prototype against the design requirements discussed in previous chapters. Below is a list of evaluation metrics:

- The system should provide a mechanism to exchange and synchronize EHRs between distributed providers using Blockchain and the cloud store.
- The system should provide patients with a secure mechanism to recover access to their EHRs.
- The system should ensure unique mapping between patients' identities and their respective EHRs.

The exchange and synchronization of EHRs between distributed providers was achieved using Ethereum public Blockchain and MS Azure Files. This was validated by accessing the EHRs of the same patient from two SDHCARE providers using the patient's fingerprint (after granting access to each provider). Figure 37 shows access results from the two SDHCARE providers.



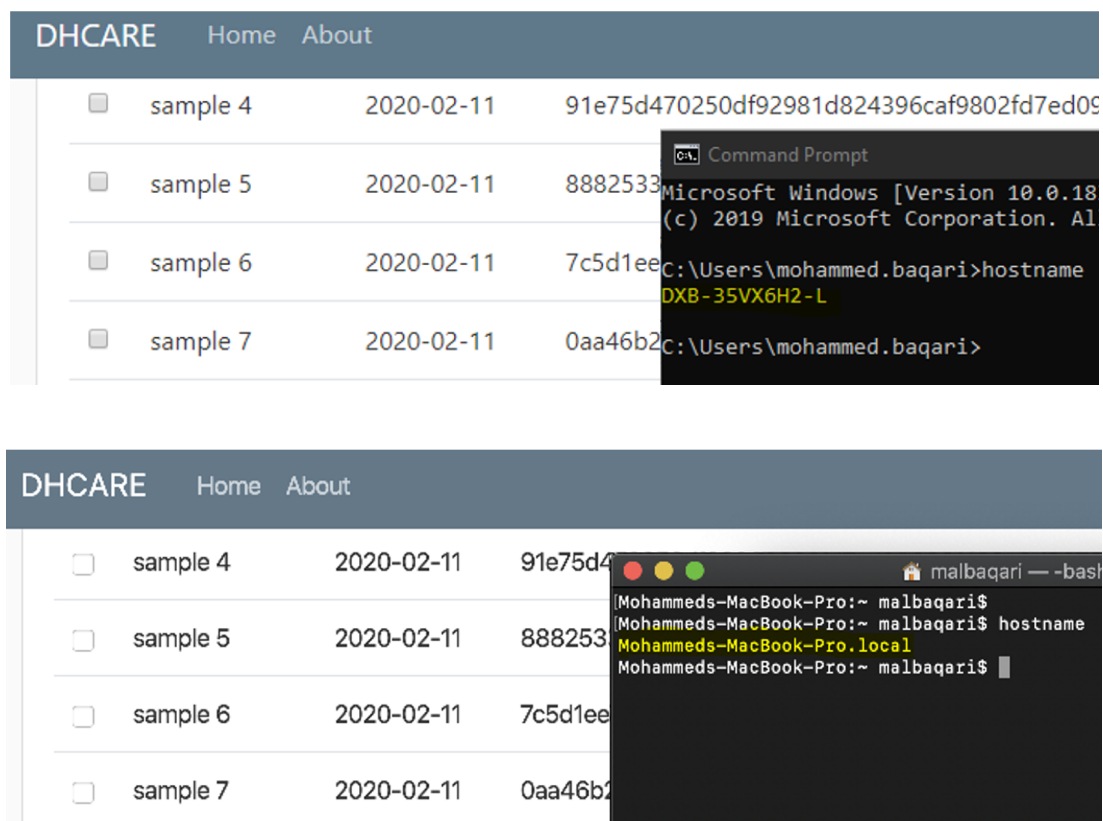


Figure 37: Synchronization of Patient's EHR

To validate the access recovery mechanism for patients' EHRs, fingerprints hashes were used as index keys to retrieve the EHR List from ehrHashTable. Figure 38 shows the requirement to provide a fingerprint in the Doctor UI to retrieve EHRs.

The screenshot shows a web application interface with a header bar containing 'DHCARE', 'Home', 'About', 'Logout', and 'Doctor'. Below the header is a form titled 'Patient Records'. The form contains a text input field labeled 'Patient Fingerprint\*' with a red border. Below the input field is a red error message: 'This field is required.' At the bottom left of the form is a button labeled 'Get Records'.

Figure 38: Verifying EHR Recovery Using Patient ID

The unique mapping between patients' fingerprints and their EHRs was validated by comparing the Merkle root hash values in ehrHashTable of the patient against the files' names stored in MS Azure Files. Figure 39 shows a sample verification.

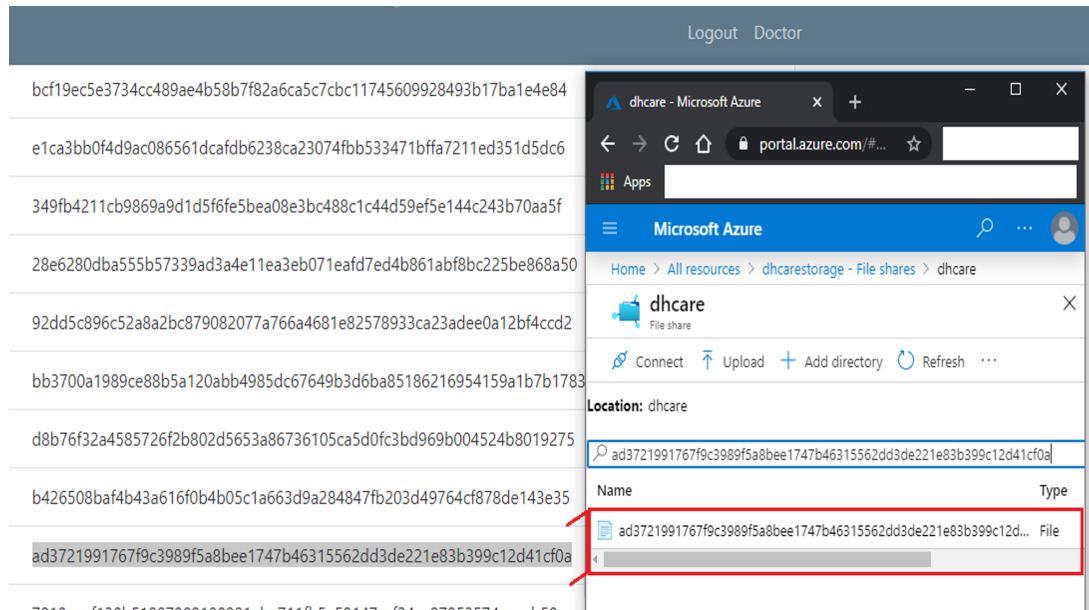


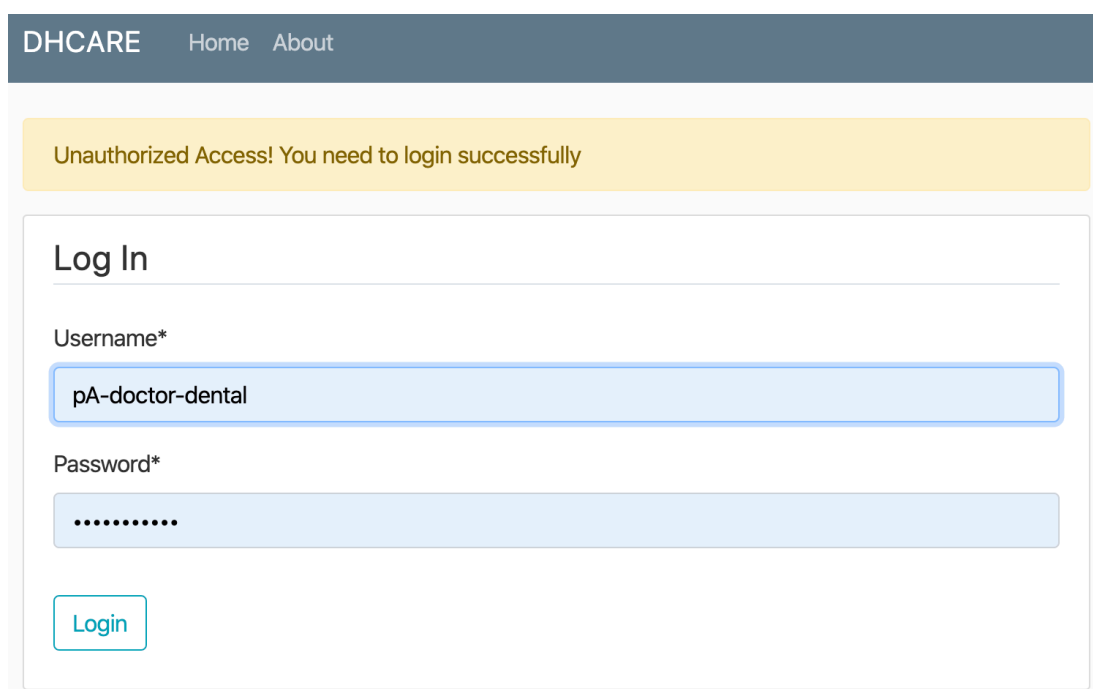
Figure 39: Verifying Hash as Names in Azure Files

## 4.2 Security Testing

The security testing of SDHCARE prototype covered the following aspects, which were discussed in previous chapters:

- The system should provide an access control mechanism to ensure authorized access to EHRs.
- The system should log all read/write activities on EHRs.
- The system should provide anonymity of EHRs in the cloud store.
- The system should validate the integrity of EHRs for read requests.

The access control system in SDHCARE was implemented at multiple levels. The first level of authorization was implemented in Django directory services to validate the group membership of the user. This ensured that only authorized users can access their role-specific UIs. Figure 40 shows a blocked attempt from a doctor attempting to access Reception UI.



The screenshot displays the DHCCARE web application interface. At the top, a dark blue header bar contains the text "DHCCARE" followed by "Home" and "About" links. Below the header, a yellow banner displays the message "Unauthorized Access! You need to login successfully". Underneath the banner is a white login box with the title "Log In". Inside the box, there are two input fields: "Username\*" with the value "pA-doctor-dental" and "Password\*" with masked characters ".....". A blue "Login" button is positioned at the bottom left of the login box.

Figure 40: Failed Login to Reception Portal using a Doctor Account

The next level of authorization was implemented using patients' fingerprints to grant doctors read or write access to EHRs. Unless a valid fingerprint is submitted by the patient, clinic doctors cannot access EHRs. Figures 41, 42, and 43 show a failed attempt to access a patient's EHR.

DHCARE

[Home](#)

[About](#)

## Doctor Portal

---

Patient Name\*

Mohammed Al Al Baqari

This field is required.

Fingerprint\*

12b2bbb1q7abjgu1v1233njl

This field is required.

Record Name\*

Sample Record

This field is required.

Date\*

11/11/2020

This field is required.

Records\*

Sample Record

Figure 41: Fingerprint Validation before Writing an EHR

DHCARE

[Home](#)

[About](#)

## Doctor Portal

---

You are unauthorized to access or modify records for this patient

Figure 42: Failed Attempt to Write a New EHR

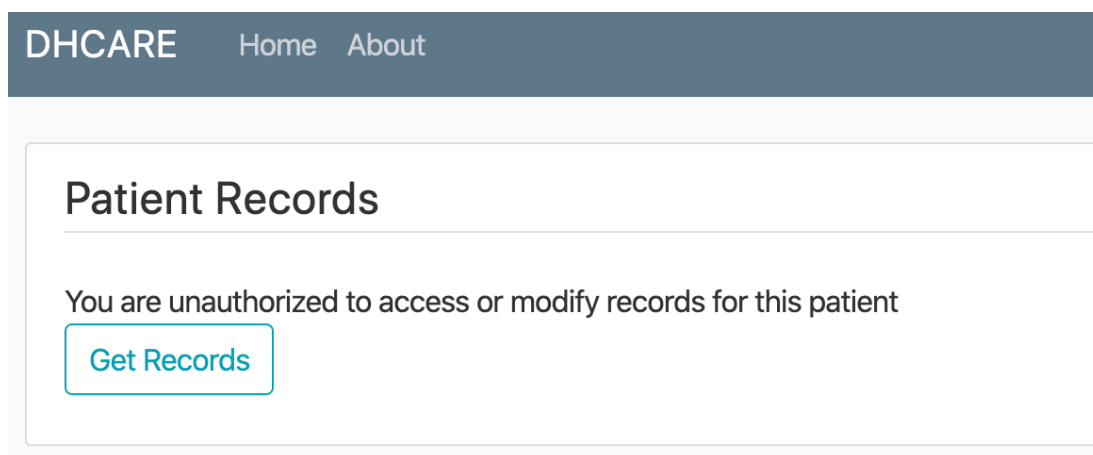



Figure 43: Failed Attempt to Read a Patient's HER


All EHR read/write activities are logged in Ethereum Blockchain for audit trace purposes. The log messages include a unique hash identifier, the Ethereum address of the healthcare provider, a timestamp, and activity details. Figure 44 shows an example from EtherScan.



All Filters

Ropsten Testnet Network

Transaction Details

Sponsored:  Fortmatic - Build Ethereum web3 dApps without browser extensions or mobile wallets. [Get Star](#)

[Overview](#)
[State Changes](#)

[ This is a Ropsten Testnet transaction only ]

Transaction Hash:	0x37366903ccfad92f9c6f0cd6856f357c4ca542af1322e0b6e49d2
Status:	Success
Block:	7400676 172 Block Confirmations
Timestamp:	12 mins ago (Feb-25-2020 03:13:39 PM +UTC)
From:	0x4df59ba9e77816a8d73f40e617b0421be333da79
To:	Contract 0xd1b73bd5256a30af32664d135618a41975f80823
Value:	0 Ether (\$0.00)
Transaction Fee:	0.000156438 Ether (\$0.000000)
Gas Limit:	156,438
Gas Used by Transaction:	156,438 (100%)

Figure 44: Sample Audit Logs for New EHR

The data stored in Azure Files were anonymized using Merkle hash values as EHR names and suppressing patients' PII. This ensured that patients' identities are not traceable from the EHR raw data. Figure 45 shows an example of anonymized data stored in MS Azure Files from the SDHCARE prototype.

h resources, services, and docs (G+/)

1

?

mohammed.albaqari@o...  
DEFAULT DIRECTORY

File shares > dhcare

Connect

Upload

Add directory

Refresh

Delete share

Edit quota

Location: dhcare

Search files by prefix

Name			
28e6280dba555b57339ad3a4e11ea3eb071eafd7ed4b861abf8bc225be8			
349fb4211cb9869a9d1d5f6fe5bea08e3bc488c1c44d59ef5e144c243b70			
7918aeaf130b51907909100991ebe711fb5a59147ccf24ce97953574aace			
92dd5c896c52a8a2bc879082077a766a4681e82578933ca23adee0a12bf			
ab00b75d014a2dc7c86f55e177ac895a2c44469e4345ddd9d73a71fb838			
ad3721991767f9c3989f5a8bee1747b46315562dd3de221e83b399c12d4			
b426508baf4b43a616f0b4b05c1a663d9a284847fb203d49764cf878de143e35	File	8.16 KiB	...
bb3700a1989ce88b5a120abb4985dc67649b3d6ba85186216954159a1b7b1783	File	8.15 KiB	...

### 3.4 System Testing

This section will cover the evaluation of DHCARE prototype implementation from functional, security and performance perspectives. The functional evaluation will ensure that the prototype is operating as expected by the design. The security evaluation is going to validate the security measures implemented in the prototype to protect it against unauthorized access. The performance evaluation will measure the elapsed time required to gain read or write access to EHRs.

#### 3.4.1 Functional Testing

The functional testing will evaluate DHCARE prototype against the requirements and the design discussed in previous chapters. Below is a list of evaluation metrics:

- â€¢ The system should provide a mechanism to exchange and synchronize EHRs between distributed providers using Blockchain
- â€¢ The system should provide patients with access recovery mechanism to their EHRs
- â€¢ The system should ensure unique mapping between patientsâ€™ identities and their respective EHRs

#### 3.4.2 Security Testing

The security testing of DHCARE prototype is going to cover the following aspects, that were

Figure 45: Sample of Anonymized Data Store in Azure Files

### 4.3 Performance Evaluation

Among all the modules in the DHCARE design, blockchain is considered the slowest component, compared to the processing speed of the other modules. This slowness is caused by the PoW consensus algorithm used in Ethereum. Hence it was the focus for performance evaluation. The time delay introduced by the blockchain layer was evaluated by validating access requests and the granting of access to EHRs. These two components are controlled by accessControlSC smart contract and patientsSC smart contract. To isolate the impact of copying speed of EHRs to the cloud and obtain accurate performance measures for the blockchain, the EHR test samples used small text files (< 20 KB). For the write test, a test patient was created and sample EHRs written into the patient's respective ehrHashTable. For the read test, each EHR sample stored in the ehrHashTable was read. In total, 15 samples were collected for read and write without/with accessControlSC smart contract.

The time delay between requests and responses was measured using Google Chrome Developer Tools. Figure 46 shows a sample time delay measurement.



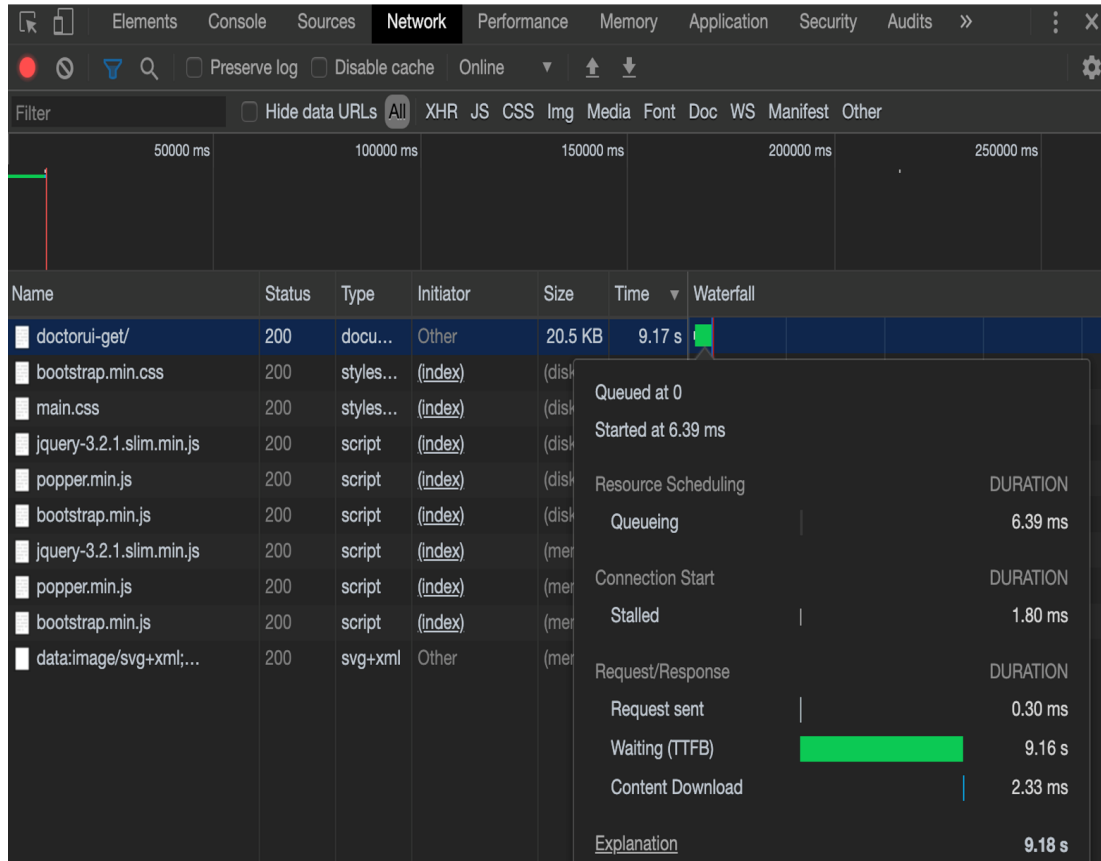


Figure 46: Performance Measurement using Google Chrome

All the evaluation test cases were executed using the same internet line to connect to Ethereum Blockchain and Azure Files. Figures 47 and 48 summarize the performance results.

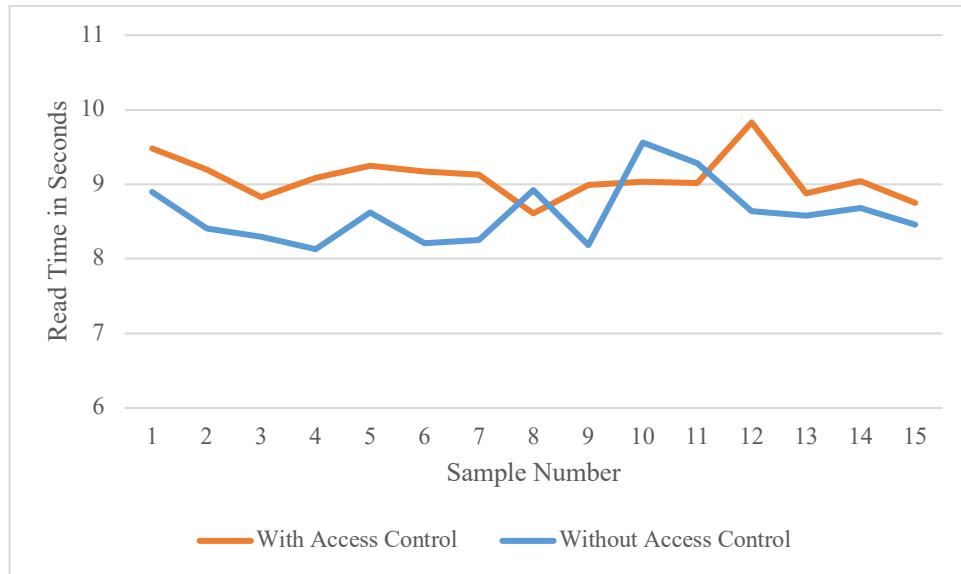


Figure 47: Read Performance Testing

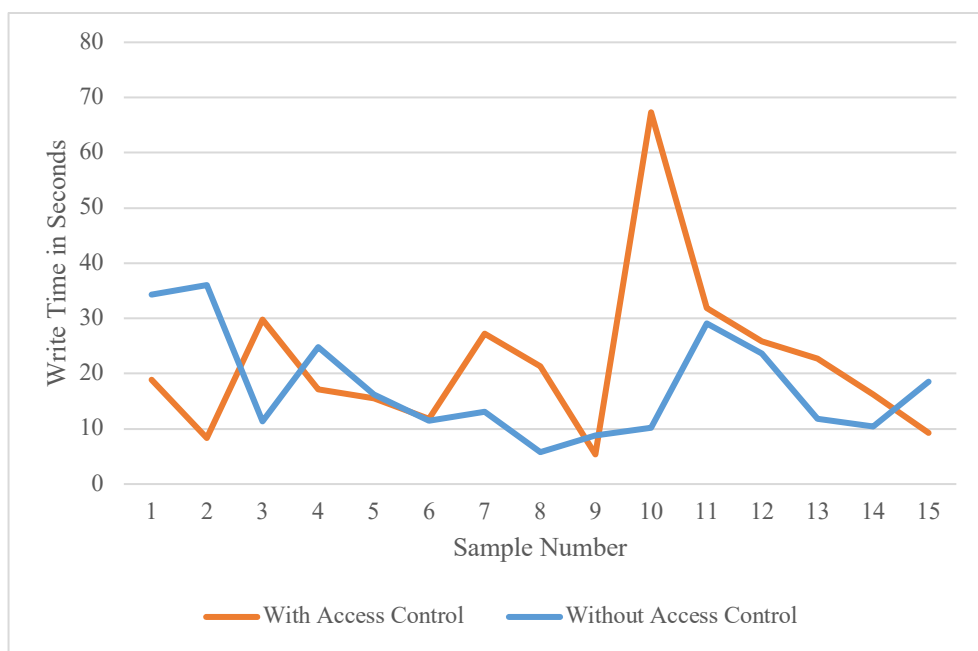


Figure 48: Write Performance Testing

Read performance was superior to the write performance. The average read time was 8.81 seconds while the average write time was 16 seconds, i.e. approximately twice the read time. This observation held with and without access control. In terms of

today's Internet speeds, the read and write times are considered low performance. However, in the real-life circumstances of most healthcare environments such time delays are acceptable. The main reason for these delays is the consensus algorithm used by Ethereum blockchain to validate and accept blocks. Another important factor is the load of the mining pool and its incentive to mine the block.

From the results, it can be concluded that no significant overhead delay is added by implementing access control in Blockchain using `accessControlSC` smart contract. This is because a single block includes thousands of transactions, and transactions from both `accessContractSC` and `paitentsSC` are usually mined in a single block (the decision to group the transactions in blocks is subject to the miner). Hence, there is no difference between sending two transactions or one transaction as they are mined in the same block. In one EHR sample (sample 6 in the write test with access control), the transactions were mined in two separate blocks. Hence, the time delay for writing the EHR metadata in `ehrHashTable` was 67.33 seconds.

Another important observation from Figure 49 is that the write time was consistent and independent of the number of records in `ehrHashTable`, while the read time was dependent on the number of records in `ehrHashTable`. In this test, five more EHR samples were added to the test patient's `ehrHashTable` (total increased to 20 samples). After re-running the read evaluation, the average read time increased from 8.81 seconds to 13.67 seconds. This is due to additional iterations executed in the code needed to list all EHR metadata associated with the patient in `ehrHashTable`. Code optimization would be required in a real-life implementation to improve the read time.

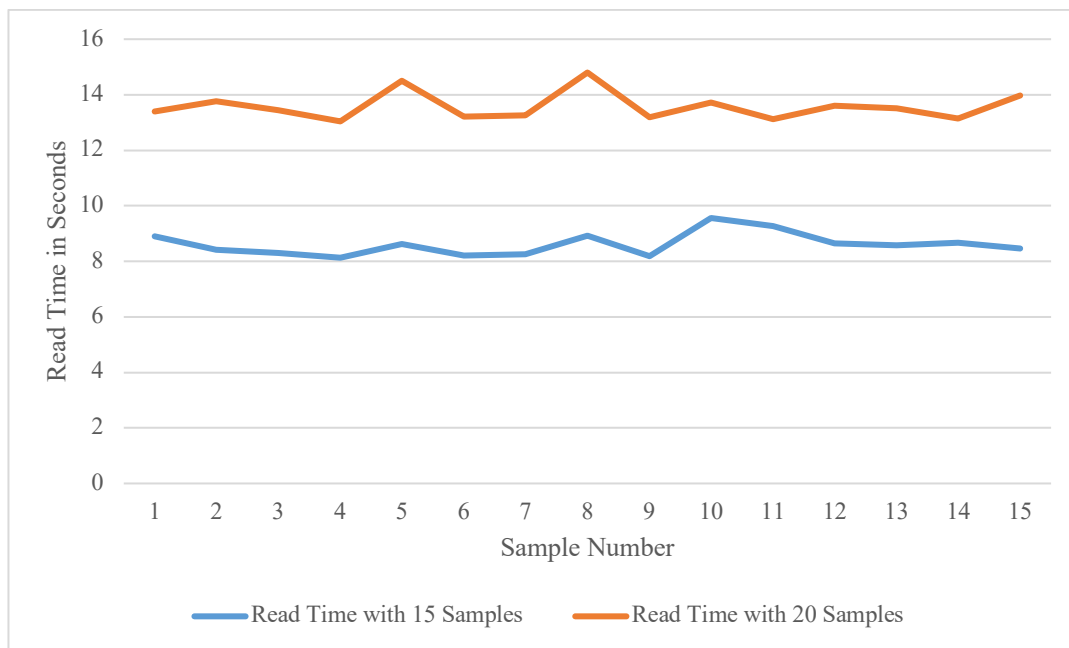


Figure 49: Read Time Analysis with Larger ehrHashTable

## Chapter 5: Conclusion

This research investigated access control recovery mechanisms for EHRs synchronized and exchanged between distributed healthcare providers using blockchain. The research project first reviewed the current state of research on blockchain in healthcare to gain understanding about the active areas. This was followed by narrowing the focus to research targeting blockchain in EHR systems.

An analysis was undertaken of current challenges in blockchain-based EHR systems and the requirements to achieve successful access control recovery mechanism for EHRs. Accordingly, the researcher proposed SDHCARE, a multilayer system that splits the roles between healthcare providers, blockchain, and a cloud store. This model system should be able to recover access to EHRs from any provider within the blockchain network. Additionally, the model may accelerate the migration of healthcare providers to blockchain-based systems through the availability of external UI integration with existing legacy healthcare environments.

A prototype was built to validate the proposed approach using Django, Python, Ethereum, and MS Azure. The prototype was coded to simulate all functional requirements and integrate the distributed layers of the design. This was followed by system validation and testing for functional requirements, security requirements, and performance. The results indicated successful operation of the proposed design from a functional and security perspective. The performance of the prototype was slow due to the functional operation of the Ethereum blockchain. However, this latency may be tolerable in healthcare environments.

For future work, the researcher will evaluate SDHCARE design against hybrid blockchain ledgers that use faster consensus algorithms. This will aim to enhance the performance of SDHCARE for EHRs read/write while maintaining the extended accessibility to the solution. Additionally, the prototype will be upgraded to use advanced biometrics combining multiple fingerprints for more accuracy and privacy, and the results should be evaluated against performance overhead. Another planned enhancement in SDHCARE will be to introduce additional roles in access control smart contract, including access delegation, access revocation, and record deletion. Finally, the researcher will evaluate the use of mobile-based biometric scanning to extend patients' manageability of access rights to EHRs.

## References

- [1] Office of the National Coordinator for Health Information Technology (ONC), “What is an electronic health record (EHR)?” 2019. [Online]. Available: <https://www.healthit.gov/faq/what-electronic-health-record-ehr>, Accessed on: 13 Nov. 2019.
  
- [2] Office of the National Coordinator for Health Information Technology (ONC) “Federal Health IT Strategic Plan 2015-2020,” 2014. [Online]. Available: <http://www.healthit.gov/sites/default/files/federal-healthIT-strategic-plan-2014.pdf>, Accessed on: 13 Nov. 2019.
  
- [3] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>, Accessed on: 21 Nov. 2019.
  
- [4] V. Buterin, “A Next-Generation Smart Contract and Decentralized Application Platform,” 2013. [Online]. Available: <https://github.com/ethereum/wiki/wiki/White-Paper>, Accessed on: 21 Nov. 2019.
  
- [5] T. McGhin, K. R. Choo, C. Z. Liu, and D. He, “Blockchain in healthcare applications: Research challenges and opportunities,” *Journal of Network and Computer Applications*, vol. 135, pp. 62–75, Jun. 2019. doi: 10.1016/j.jnca.2019.02.027

- [6] National Institute of Standardization and Technology (NIST), Computer Security Resource Center, “Glossary” 2019. [Online]. Available: <https://csrc.nist.gov/Glossary>, Accessed on: 11 Nov. 2019.
- [7] A. P. Joshi, M. Han, and Y. Wang, “A survey on security and privacy issues of blockchain technology,” *Mathematical Foundations of Computing*, vol. 1, no. 2, pp. 121–147, May 2018. doi: 10.3934/mfc.2018007
- [8] I. Weber, V. Gramoli, A. Ponomarev, M. Staples, R. Holz, A. B. Tran, and P. Rimba, “On Availability for Blockchain-Based Systems,” *36th IEEE Symposium on Reliable Distributed Systems (SRDS 2017)*, Hong Kong, China, 2017, pp. 64–73. doi: 10.1109/SRDS.2017.15
- [9] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain Technology Overview” (NISTIR 8202), 2018. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf>, Accessed on: 27 Oct. 2019.
- [10] H. Halpin and M. Piekarska, “Introduction to Security and Privacy on the Blockchain,” *2nd IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Paris, France, 2017, pp. 1–3. doi: 10.1109/EuroSPW.2017.43 [Online]. Available: <https://hal.inria.fr/hal-01673293/document>, Accessed on: 17 Oct. 2019
- [11] N. Popper, “A Hacking of More Than \$50 Million Dashes Hopes in the World of Virtual Currency,” *New York Times*, Jun. 2016. [Online]. <https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have->



removed-more-than-50-million-from-experimental-cybercurrency-project.html, Accessed on: 27 Oct. 2019

- [12] P. Zhong, Q. Zhong, H. Mi, S. Zhang, and Y. Xiang, "Privacy-Protected Blockchain System," *First International Workshop on Blockchain and Mobile Applications (BlockApp'19 Workshop)*, in conjunction with 20th IEEE International Conference on Mobile Data Management (MDM 2019), Hong Kong, China, 2019, pp. 457–461. doi: 2019. 10.1109/MDM.2019.000-2
- [13] M. Cash and M. Bassiouni, "Two-Tier Permission-ed and Permission-Less Blockchain for Secure Data Sharing," 3rd IEEE International Conference on Smart Cloud (SmartCloud 2018), New York, USA, 2018. doi: 10.1109/SmartCloud.2018.00031
- [14] D. K. Tosh, S. Shetty, X. Liang, C. A. Kamhoua, K. A. Kwiat, and L. Njilla, "Security Implications of Blockchain Cloud with Analysis of Block Withholding Attack," *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Madrid, Spain, 2017, pp. 458–467. doi: 10.1109/CCGRID.2017.111 [abstract]
- [15] G. Magyar, "Blockchain: solving the privacy and research availability tradeoff for EHR data: A new disruptive technology in health data management," *IEEE 30th Jubilee Neumann Colloquium (NC)*, Budapest, Hungary, 2017, pp. 135–140. doi: 10.1109/NC.2017.8263269
- [16] M. Pilkington, "Can Blockchain Improve Healthcare Management? Consumer Medical Electronics and the IoMT," 2017. [Online]. Available: <https://ssrn.com/abstract=3025393>, Accessed on: 8 Dec. 2019.

- [17] P. Zhang, M. A. Walker, J. White, D. C. Schmidt, and G. Lenz, “Metrics for assessing blockchain-based healthcare decentralized apps,” *IEEE 19th International Conference on e-Health Networking, Applications & Services (Healthcom 2017)*, Dalian, China, 2017. doi: 10.1109/HealthCom.2017.8210842
- [18] G. G. Dagher, J. Mohler, M. Milojkovic, and P. B. Marella, “Ancile: Privacy-preserving framework for access control and interoperability of electronic health records using blockchain technology,” *Sustainable Cities and Society*, vol. 39, pp. 283–297, May 2018. doi: 10.1016/j.scs.2018.02.014
- [19] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, “MedRec: Using Blockchain for Medical Data Access and Permission Management,” *2nd International Conference on Open and Big Data (OBD 2016)*, Vienna, Austria, 2016, pp. 25–30. doi: 10.1109/OBD.2016.11
- [20] Q. Xia, E. B. Sifah, A. Smahi, S. Amofa and X. Zhang, “BBDS: Blockchain-Based Data Sharing for Electronic Medical Records in Cloud Environments,” *Information*, vol. 8, no. 2, Apr. 2017, Art. no. 44. doi: 10.3390/info8020044
- [21] Y. Yang, X. Li, N. Qamar, P. Liu, W. Ke, B. Shen, and Z. Liu, “Medshare: A Novel Hybrid Cloud for Medical Resource Sharing Among Autonomous Healthcare Providers,” *IEEE Access*, vol. 6, pp. 46949–46961, Aug. 2018. doi: 10.1109/ACCESS.2018.2865535
- [22] A. Roehrs, C. A. da Costa, and R. da Rosa Righi, “OmniPHR: A distributed architecture model to integrate personal health records,” *Journal of*

*Biomedical Informatics*, vol. 71, pp. 70–81, Jul. 2017.  
doi.org/10.1016/j.jbi.2017.05.012

- [23] K. Fan, S. Wang, Y. Ren, H. Li, and Y. Yang, “MedBlock: Efficient and Secure Medical Data Sharing Via Blockchain,” *Journal of Medical Systems*, vol. 42, article 136, Jun. 2018. doi: 10.1007/s10916-018-0993-7
- [24] Microsoft Corp., Microsoft Azure, “What is Azure Files?” 2020. [Online]. Available: <https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction>, Accessed on: 31 Jan. 2020.

## Appendix

This section will cover the low-level coding of DHCARE Django and Ethereum components.

### Appendix A – Ethereum Smart Contracts

Smart Contract dnsSC

```
pragma solidity ^0.6.1;

contract dnsSC{

    struct provider{
        string name;
        string webAddress;
    }

    mapping (address => provider) providersTable;

    function createProvider (string memory _name, string memory
    _webAddress) public {
        address OHP = msg.sender;
        providersTable[OHP] = provider (_name, _webAddress);
    }

    function getProvider (address _OHP) public view returns (string memory
    _name, string memory _webAddress) {
        _name = providersTable[_OHP].name;
        _webAddress = providersTable[_OHP].webAddress;
    }

}
```

Smart Contract patientsSC

```
pragma solidity ^0.6.1;

contract patientsSC{

    struct patient{
        string OHP;
        string receptionID;
    }

    struct ehr{
        string name;
        string hash;
        string status;
        string date;
    }

    mapping (uint => patient) patientsTable;
    mapping (uint => ehr[]) ehrHashTable;

    function createPatient (uint _fingerprint, string memory _OHP, string
    memory _receptionID) public {
```

```

        patientsTable[_fingerprint] = patient (_OHP, _receptionID);
    }

    function getPatient (uint _fingerprint) public view returns (string
memory _OHP, string memory _receptionID) {
        _OHP = patientsTable[_fingerprint].OHP;
        _receptionID = patientsTable[_fingerprint].receptionID;
    }

    function createEhr (uint _fingerprint, string memory _name, string
memory _hash, string memory _status, string memory _date) public {
        ehrHashTable[_fingerprint].push (ehr(_name, _hash, _status,
_date));
    }

    function getEhr (uint _fingerprint, uint _count) public view returns
(string memory _name, string memory _hash, string memory _status, string
memory _date) {
        _name = ehrHashTable[_fingerprint][_count].name;
        _hash = ehrHashTable[_fingerprint][_count].hash;
        _status = ehrHashTable[_fingerprint][_count].status;
        _date = ehrHashTable[_fingerprint][_count].date;
    }
}

```

Smart Contract accessControlSC

```

pragma solidity ^0.6.1;

contract accessControlSC{

    mapping (string => string[]) PACL;

    function addClinic (string memory _fingerprint, string memory
_clinicID) public {
        PACL[_fingerprint].push (_clinicID);
    }

    function grantClinicAccess (string memory _fingerprint, string memory
_clinicID) public view returns (string memory) {

        uint i;
        for (i=0; i<=PACL[_fingerprint].length; i++){
            if (keccak256(abi.encodePacked((PACL[_fingerprint][i]))) ==
keccak256(abi.encodePacked((_clinicID)))){
                return 'true';
            }
        }
        return 'false';
    }
}

```

## Appendix B – Ethereum Smart Contract ABIs

accessControlSC\_abi.json

```

[
  {
    "inputs": [
      {
        "internalType": "string",
        "name": "_fingerprint",

```

```

        "type": "string"
      },
      {
        "internalType": "string",
        "name": "_clinicID",
        "type": "string"
      }
    ],
    "name": "addClinic",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "inputs": [
      {
        "internalType": "string",
        "name": "_fingerprint",
        "type": "string"
      },
      {
        "internalType": "string",
        "name": "_clinicID",
        "type": "string"
      }
    ],
    "name": "grantClinicAccess",
    "outputs": [
      {
        "internalType": "string",
        "name": "",
        "type": "string"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  }
]

```

patientsSC\_abi.json

```

[
  {
    "inputs": [
      {
        "internalType": "uint256",
        "name": "_fingerprint",
        "type": "uint256"
      },
      {
        "internalType": "string",
        "name": "_name",
        "type": "string"
      },
      {
        "internalType": "string",
        "name": "_hash",
        "type": "string"
      },
      {
        "internalType": "string",
        "name": "_status",
        "type": "string"
      }
    ]
  }
]

```

```

    },
    {
      "internalType": "string",
      "name": "_date",
      "type": "string"
    }
  ],
  "name": "createEhr",
  "outputs": [],
  "stateMutability": "nonpayable",
  "type": "function"
},
{
  "inputs": [
    {
      "internalType": "uint256",
      "name": "_fingerprint",
      "type": "uint256"
    },
    {
      "internalType": "string",
      "name": "_OHP",
      "type": "string"
    },
    {
      "internalType": "string",
      "name": "_receptionID",
      "type": "string"
    }
  ],
  "name": "createPatient",
  "outputs": [],
  "stateMutability": "nonpayable",
  "type": "function"
},
{
  "inputs": [
    {
      "internalType": "uint256",
      "name": "_fingerprint",
      "type": "uint256"
    },
    {
      "internalType": "uint256",
      "name": "_count",
      "type": "uint256"
    }
  ],
  "name": "getEhr",
  "outputs": [
    {
      "internalType": "string",
      "name": "_name",
      "type": "string"
    },
    {
      "internalType": "string",
      "name": "_hash",
      "type": "string"
    },
    {
      "internalType": "string",
      "name": "_status",
      "type": "string"
    }
  ],

```

```

        {
            "internalType": "string",
            "name": "_date",
            "type": "string"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [
        {
            "internalType": "uint256",
            "name": "_fingerprint",
            "type": "uint256"
        }
    ],
    "name": "getPatient",
    "outputs": [
        {
            "internalType": "string",
            "name": "_OHP",
            "type": "string"
        },
        {
            "internalType": "string",
            "name": "_receptionID",
            "type": "string"
        }
    ],
    "stateMutability": "view",
    "type": "function"
}
]

```

dnsSC\_abi.json

```

[
    {
        "inputs": [
            {
                "internalType": "string",
                "name": "_name",
                "type": "string"
            },
            {
                "internalType": "string",
                "name": "_webAddress",
                "type": "string"
            }
        ],
        "name": "createProvider",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [
            {
                "internalType": "address",
                "name": "_OHP",
                "type": "address"
            }
        ]
    }
]

```



```

    ],
    "name": "getProvider",
    "outputs": [
        {
            "internalType": "string",
            "name": "_name",
            "type": "string"
        },
        {
            "internalType": "string",
            "name": "_webAddress",
            "type": "string"
        }
    ]
},
    "stateMutability": "view",
    "type": "function"
}
]

```

## Appendix C – Django Main Application Code

urls.py

```

from dhcare import views as dhcare_views
from django.contrib import admin
from django.contrib.auth import views as auth_views
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('adminui/', dhcare_views.adminui, name='adminui'),
    path('receptionui/', dhcare_views.receptionui, name='receptionui'),
    path('receptionui-get/', dhcare_views.receptionui_get,
name='receptionui-get'),
    path('receptionui-book/', dhcare_views.receptionui_book,
name='receptionui-book'),
    path('appointment-confirm/', dhcare_views.appointment_confirm,
name='appointment-confirm'),
    path('doctorui-get/', dhcare_views.doctorui_get, name='doctorui-get'),
    path('doctorui-submit/', dhcare_views.doctorui_submit, name='doctorui-
submit'),
    path('doctorui/', dhcare_views.doctorui, name='doctorui'),
    path('new-patient/', dhcare_views.new_patient, name='new-patient'),
    path('adminui-get/', dhcare_views.adminui_get, name='adminui_get'),
    path('adminui-submit/', dhcare_views.adminui_submit,
name='adminui_submit'),
    path('login/',
auth_views.LoginView.as_view(template_name='dhcare/login.html'),
name='login'),
    path('logout/',
auth_views.LogoutView.as_view(template_name='dhcare/logout.html'),
name='logout'),
    path('', include('dhcare.urls')),
]

```

settings.py

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['*']
# Application definition

INSTALLED_APPS = [
    'dhcare.apps.DhcareConfig',
    'crispy_forms',
    'django_tables2',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

STATIC_URL = '/static/'

CRISPY_TEMPLATE_PACK = 'bootstrap4'

LOGIN_REDIRECT_URL = 'dhcare-home'
LOGIN_URL = 'login'
```

## Appendix D – Django DHCARE Application Code

accessControlSC.py

```
import json

from web3 import Web3

def accessControlSC_add(fingerprint, clinic_id):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/accessControlSC_abi.json') as json_file:
        accessControlSC_abi = json.load(json_file)

    accessControlSC_address = '0x1b3135e3Cd2Ee573A2caC760561A15DB22ac92A3'

    accessControlSC = web3.eth.contract(address=accessControlSC_address,
abi=accessControlSC_abi)
    key =
'0x3E67D814F4794E3172A94C8AF582C75A6B4868F4FAC912F234E78AE97D44518A'
    acct = web3.eth.account.privateKeyToAccount(key)
    account_address = acct.address

    try:
        tx = accessControlSC.functions.addClinic(fingerprint,
clinic_id).buildTransaction(
            {'nonce': web3.eth.getTransactionCount(account_address)})
        signed_tx = web3.eth.account.signTransaction(tx, key)
        hash = web3.eth.sendRawTransaction(signed_tx.rawTransaction)

    tx_receipt=web3.eth.waitForTransactionReceipt(hash).get('transactionHash').
hex()

    return ('The appointment has been confirmed and access granted to
the clinic. Your Transaction Ref. is '
```

```

        + str(tx_receipt) + ' Please remember it for tracking
purpose')
    except:
        return ('Unable to Submit Information')

def accessControlSC_check(fingerprint, clinic_id):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/accessControlSC_abi.json') as json_file:
        accessControlSC_abi = json.load(json_file)

    accessControlSC_address = '0x1b3135e3Cd2Ee573A2caC760561A15DB22ac92A3'

    accessControlSC = web3.eth.contract(address=accessControlSC_address,
abi=accessControlSC_abi)

    try:
        accessRequest =
accessControlSC.functions.grantClinicAccess(fingerprint, clinic_id).call()
        return (accessRequest)
    except:
        return ('You are unauthorized to access or modify records for this
patient')

```

### patientsSC.py

```

import json

from web3 import Web3

def patientsSC_patient_post(fingerprint, ohp, reception_id):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/patientsSC_abi.json') as json_file:
        patientsSC_abi = json.load(json_file)

    patientsSC_address = '0xD1b73bd5256a30Af32664D135618A41975f80823'

    patientsSC = web3.eth.contract(address=patientsSC_address,
abi=patientsSC_abi)
    key =
'0x3E67D814F4794E3172A94C8AF582C75A6B4868F4FAC912F234E78AE97D44518A'
    acct = web3.eth.account.privateKeyToAccount(key)
    account_address = acct.address

    try:
        tx = patientsSC.functions.createPatient(fingerprint, ohp,
reception_id).buildTransaction(
            {'nonce': web3.eth.getTransactionCount(account_address)})
        signed_tx = web3.eth.account.signTransaction(tx, key)
        hash = web3.eth.sendRawTransaction(signed_tx.rawTransaction)

    tx_receipt=web3.eth.waitForTransactionReceipt(hash).get('transactionHash').
hex()
        return ('Patient File was Created Successfully! Transaction Ref. '
+ str(tx_receipt))
    except:
        return ('Unable to Submit Information')

```

```

def patientsSC_ehr_post(fingerprint, name, hash, status, date):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/patientsSC_abi.json') as json_file:
        patientsSC_abi = json.load(json_file)

    patientsSC_address = '0xD1b73bd5256a30Af32664D135618A41975f80823'

    patientsSC = web3.eth.contract(address=patientsSC_address,
abi=patientsSC_abi)
    key =
'0x3E67D814F4794E3172A94C8AF582C75A6B4868F4FAC912F234E78AE97D44518A'
    acct = web3.eth.account.privateKeyToAccount(key)
    account_address = acct.address

    try:
        tx = patientsSC.functions.createEhr(fingerprint, name, hash,
status, date).buildTransaction(
            {'nonce': web3.eth.getTransactionCount(account_address)})
        signed_tx = web3.eth.account.signTransaction(tx, key)
        hash = web3.eth.sendRawTransaction(signed_tx.rawTransaction)

    tx_receipt=web3.eth.waitForTransactionReceipt(hash).get('transactionHash').
hex()
        return ('Patient Records Uploaded Sucessfully! Transaction Ref. ' +
str(tx_receipt))
    except:
        return ('Unable to Upload Reocrds')

def patientsSC_ehr_get(fingerprint):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/patientsSC_abi.json') as json_file:
        patientsSC_abi = json.load(json_file)

    patientsSC_address = '0xD1b73bd5256a30Af32664D135618A41975f80823'

    patientsSC = web3.eth.contract(address=patientsSC_address,
abi=patientsSC_abi)

    try:
        count = 0
        getEhr = patientsSC.functions.getEhr(fingerprint, count).call()
        output = [{'id':count, 'record_name':getEhr[0],
'record_hash':getEhr[1], 'record_status':getEhr[2],
'record_date':getEhr[3]}]
        while True:
            try:
                count = count + 1
                getEhr = patientsSC.functions.getEhr(fingerprint,
count).call()
                output.append({'id':count, 'record_name':getEhr[0],
'record_hash':getEhr[1], 'record_status':getEhr[2],
'record_date':getEhr[3]})
            except:
                return (output)
    except:
        return ('Unable to Retrieve Information')

```

## dnsSC.py

```

import json

from web3 import Web3

def dnsSC_get(account_address):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/dnsSC_abi.json') as json_file:
        dnsSC_abi = json.load(json_file)

    dnsSC_address = '0x274D835998CDb077C224531619eB6e4D86b7739d'

    dnsSC = web3.eth.contract(address=dnsSC_address, abi=dnsSC_abi)

    try:
        getProvider = dnsSC.functions.getProvider(account_address).call()
        provider = {'provider_name': getProvider[0], 'provider_webAddress':
getProvider[1]}
        return (provider)
    except:
        return ('Unable to Retrieve Information')

def dnsSC_post(name, webAddress):
    infura_url =
'https://ropsten.infura.io/v3/db1690d7911842a6a0ec7690d08a0ca3'
    web3 = Web3(Web3.HTTPProvider(infura_url))

    with open('dhcare/dnsSC_abi.json') as json_file:
        dnsSC_abi = json.load(json_file)

    dnsSC_address = '0x274D835998CDb077C224531619eB6e4D86b7739d'

    dnsSC = web3.eth.contract(address=dnsSC_address, abi=dnsSC_abi)
    key =
'0x3E67D814F4794E3172A94C8AF582C75A6B4868F4FAC912F234E78AE97D44518A'
    acct = web3.eth.account.privateKeyToAccount(key)
    account_address = acct.address

    try:
        tx = dnsSC.functions.createProvider(name,
webAddress).buildTransaction(
        {'nonce': web3.eth.getTransactionCount(account_address)})
        signed_tx = web3.eth.account.signTransaction(tx, key)
        hash = web3.eth.sendRawTransaction(signed_tx.rawTransaction)

    tx_receipt=web3.eth.waitForTransactionReceipt(hash).get('transactionHash').
hex()
        return ('Information was Submitted Successfully! Transaction Ref. '
+ str(tx_receipt))
    except:
        return ('Unable to Submit Information')

```

## admin.py

```

from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.models import User

```

```

from .models import provider, department, appointment, Profile

class ProfileInline(admin.StackedInline):
    model = Profile
    can_delete = False
    verbose_name_plural = 'Profile'
    fk_name = 'user'

class CustomUserAdmin(UserAdmin):
    inlines = (ProfileInline, )

    def get_inline_instances(self, request, obj=None):
        if not obj:
            return list()
        return super(CustomUserAdmin, self).get_inline_instances(request,
obj)

admin.site.register(provider)
admin.site.register(department)
admin.site.register(appointment)
admin.site.unregister(User)
admin.site.register(User, CustomUserAdmin)

```

apps.py

```

from django.apps import AppConfig

class DhcareConfig(AppConfig):
    name = 'dhcare'

```

azure\_files.py

```

from azure.storage.file import FileService
from azure.storage.file import ContentSettings

file_service = FileService(account_name='dhcarestorage',

account_key='ArdixcQfhaAfmwC9XueTPaJYXEDXa3CXlXoYZ7Z76GigvDWJz4WDDqmxQtB61q
IPk+4rr+r71WXEQX80ruIsQ==')

# file_service.create_share('dhcare')

# def get_file():
#     name = ''
#     value = ''
#     file_service.get_file_to_text('dhcare', None, name, encoding='utf-8')
#     return (name)

def create_ehr(name, value):
    file_service.create_file_from_text(
        'dhcare',
        None, # We want to create this blob in the root directory, so we
specify None for the directory_name
        name,
        value,

```

```
encoding = 'utf-8',
content_settings = ContentSettings(content_type='txt'))
```

### custom\_decorators.py

```
from functools import wraps
from urllib.parse import urlparse

from django.conf import settings
from django.contrib import messages
from django.contrib.auth import REDIRECT_FIELD_NAME
from django.shortcuts import resolve_url

def custom_user_passes_test(test_func, login_url=None,
                             redirect_field_name=REDIRECT_FIELD_NAME):
    """
    Decorator for views that checks that the user passes the given test,
    redirecting to the log-in page if necessary. The test should be a
    callable
    that takes the user object and returns True if the user passes.
    """

    def decorator(view_func):
        @wraps(view_func)
        def _wrapped_view(request, *args, **kwargs):
            if test_func(request.user):
                return view_func(request, *args, **kwargs)
            path = request.build_absolute_uri()
            resolved_login_url = resolve_url(login_url or
settings.LOGIN_URL)
            # If the login url is the same scheme and net location then
just
            # use the path as the "next" url.
            login_scheme, login_netloc = urlparse(resolved_login_url)[:2]
            current_scheme, current_netloc = urlparse(path)[:2]
            if ((not login_scheme or login_scheme == current_scheme) and
                (not login_netloc or login_netloc == current_netloc)):
                path = request.get_full_path()
            from django.contrib.auth.views import redirect_to_login
            messages.warning(request, 'Unauthorized Access! You need to
login successfully')
            return redirect_to_login(
                path, resolved_login_url, redirect_field_name)

        return _wrapped_view

    return decorator
```

### forms.py

```
from django import forms

from .models import appointment

# By default DateInput takes input as string. Using this class, we can
change the input as Calender
class DateInput(forms.DateInput):
    input_type = 'date'
```

```

class TimeInput(forms.TimeInput):
    input_type = 'time'

class getProviderInfo(forms.Form):
    OHP_Eth = forms.CharField(label='Hospital Ethereum Address',
max_length=100)

class submitProviderInfo(forms.Form):
    name = forms.CharField(label='Hospital Name', max_length=100)
    webAddress = forms.CharField(label='Hospital Web Address',
max_length=100)

class bookAppointment(forms.ModelForm):
    class Meta:
        model = appointment
        fields = ['name', 'nid', 'date', 'time', 'department_code']
        labels = {
            'name': 'Patient Name',
            'nid': 'National ID',
            'date': 'Date',
            'time': 'Time',
            'department_code': 'Clinic'
        }
        widgets = {'date': DateInput(), 'time': TimeInput()}

class getAppointments(forms.Form):
    nid = forms.IntegerField(label='Patient National ID')

class newPatient(forms.Form):
    name = forms.CharField(label='Patient Name', max_length=100)
    dob = forms.DateField(label='Date of Birth', widget=DateInput)
    fingerprint = forms.CharField(label='Fingerprint', max_length=100)

class submitRecords(forms.Form):
    patient_name = forms.CharField(label='Patient Name', max_length=100)
    patient_fingerprint = forms.CharField(label='Fingerprint',
max_length=100)
    record_name = forms.CharField(label='Record Name', max_length=100)
    record_date = forms.DateField(label='Date', widget=DateInput)
    record_description = forms.CharField(label='Records',
widget=forms.Textarea)

class getRecords(forms.Form):
    patient_fingerprint = forms.CharField(label='Patient Fingerprint',
max_length=100)

```

models.py

```

from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class department(models.Model):
    code = models.IntegerField(unique=True, primary_key=True)
    name = models.CharField(max_length=100)

```



```

    # This is required to show the name attribute of the database instead
    of 'department object (code)'
    # in the drop-down
    def __str__(self):
        return self.name

class provider(models.Model):
    ohp = models.CharField(max_length=100, primary_key=True)
    secret = models.CharField(max_length=100)

    def __str__(self):
        return self.ohp

class appointment(models.Model):
    name = models.CharField(max_length=100)
    nid = models.IntegerField()
    date = models.DateField(default=timezone.now)
    time = models.TimeField(default=timezone.now)

    # This is better than 'CHOICES' because 'CHOICES' need static values
    for the drop-down but this is
    # polled from the database department
    department_code = models.ForeignKey('department',
on_delete=models.CASCADE)

    def __str__(self):
        return self.name

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    department_code = models.ForeignKey('department',
on_delete=models.CASCADE, blank=True)

@receiver(post_save, sender=User)
def create_or_update_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
    instance.profile.save()

```

## tables.py

```

import django_tables2 as tables

from .models import appointment

class appointmentTable(tables.Table):
    department_code = tables.Column(verbose_name='Clinic')
    #nid = tables.Column(verbose_name='National ID')
    # id = tables.Column(visible=False)
    id = tables.CheckBoxColumn(accessor='id')

    class Meta:
        model = appointment
        template_name = "django_tables2/bootstrap4.html"

class ehrTable(tables.Table):
    id = tables.CheckBoxColumn(accessor='id')
    record_name = tables.Column(verbose_name='Record Name',
order_by='name')
    record_date = tables.Column(verbose_name='Record Date')

```

```

record_hash = tables.Column(verbose_name=' Record Hash')
record_status = tables.Column(verbose_name='Record Status')

class Meta:
    template_name = "django_tables2/bootstrap4.html"

```

urls.py

```

from django.urls import path

from . import views

urlpatterns = [
    path('', views.home, name='dhcare-home'),
    path('about/', views.about, name='dhcare-about'),
]

```

views.py

```

import hashlib

from django.contrib import messages
from django.contrib.auth.models import Group
from django.shortcuts import render
from django.contrib.auth.models import User
from .accessControlSC import accessControlSC_add, accessControlSC_check
from .custom_decorators import custom_user_passes_test
from .dnsSC import dnsSC_get, dnsSC_post
from .forms import getProviderInfo, submitProviderInfo, getAppointments,
getRecords
from .forms import submitRecords, bookAppointment, newPatient
from .models import appointment, provider, department
from .patientsSC import patientsSC_patient_post, patientsSC_ehr_post,
patientsSC_ehr_get
from .tables import appointmentTable, ehrTable
from django_tables2 import RequestConfig
from .azure_files import create_ehr

def home(request):
    return render(request, 'dhcare/home.html')

def about(request):
    return render(request, 'dhcare/about.html', {'title': 'About'})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Admins')
in u.groups.all())
def adminui(request):
    return render(request, 'dhcare/adminui.html')

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Admins')
in u.groups.all())
def adminui_get(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
request:
        form = getProviderInfo(request.POST)

```

```

        # check whether it's valid:
        if form.is_valid():
            account_address = form.cleaned_data.get('OHP_Eth')
            provider = dnsSC_get(account_address)
            context = {
                'provider': provider
            }

            return render(request, 'dhcare/adminui-get.html', context)

    # if a GET (or any other method) we'll create a blank form
    else:
        form = getProviderInfo()

    return render(request, 'dhcare/adminui-get.html', {'form': form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Admins')
in u.groups.all())
def adminui_submit(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
        request:
        form = submitProviderInfo(request.POST)
        # check whether it's valid:
        if form.is_valid():
            providerTx = dnsSC_post(form.cleaned_data.get('name'),
form.cleaned_data.get('webAddress'))
            context = {
                'providerTx': providerTx
            }

            return render(request, 'dhcare/adminui-submit.html', context)

    # if a GET (or any other method) we'll create a blank form
    else:
        form = submitProviderInfo()

    return render(request, 'dhcare/adminui-submit.html', {'form': form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-
Reception') in u.groups.all())
def receptionui(request):
    return render(request, 'dhcare/receptionui.html')

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-
Reception') in u.groups.all())
def receptionui_get(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
        request:
        form = getAppointments(request.POST)
        # check whether it's valid:
        if form.is_valid():
            nid = form.cleaned_data.get('nid')
            table = appointmentTable(appointment.objects.filter(nid=nid))
            return render(request, 'dhcare/receptionui-get.html', {"table":
table})

    else:
        form = getAppointments()

```

```

        return render(request, 'dhcare/receptionui-get.html', {'form': form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-
Reception') in u.groups.all())
def new_patient(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
request:
        form = newPatient(request.POST)
        # check whether it's valid:
        if form.is_valid():
            provider_db = provider.objects.all()
            for item in provider_db:
                ohp = str(item.ohp)

            reception_db = department.objects.filter(name='Reception')
            for item in reception_db:
                reception_id = str(item.code)
                name = form.cleaned_data.get('name')
                dob = form.cleaned_data.get('dob')
                fingerprint = form.cleaned_data.get('fingerprint')
                confirmation = patientsSC_patient_post(int(fingerprint), name,
str(dob), ohp, reception_id)

            return render(request, 'dhcare/new-patient.html',
{'confirmation': confirmation})

        # if a GET (or any other method) we'll create a blank form
    else:
        form = newPatient()

    return render(request, 'dhcare/new-patient.html', {'form': form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-
Reception') in u.groups.all())
def receptionui_book(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
request:
        form = bookAppointment(request.POST)
        # check whether it's valid:
        if form.is_valid():
            form.save()
            patient_name = form.cleaned_data.get('name')
            messages.success(request, 'Appointment Booking Confirmed for
Patient Name {}'.format(patient_name))
            return render(request, 'dhcare/receptionui.html')

        # if a GET (or any other method) we'll create a blank form
    else:
        form = bookAppointment()

    return render(request, 'dhcare/receptionui-book.html', {'form': form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-
Reception') in u.groups.all())
def appointment_confirm(request):
    if request.method == 'POST':
        checkbox_id = request.POST.get('id')
        checkbox_table = appointment.objects.filter(id=checkbox_id)

```

```

        for item in checkbox_table:
            fingerprint = str(item.fingerprint)
            department_code = str(item.department_code)
            accessControlSC_tx = accessControlSC_add(fingerprint,
department_code)
            appointment.objects.filter(id=checkbox_id).delete()
            return render(request, 'dhcare/appointments.html',
{"accessControlSC_tx": accessControlSC_tx})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Doctors')
in u.groups.all())
def doctorui(request):
    return render(request, 'dhcare/doctorui.html')

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Doctors')
in u.groups.all())
def doctorui_get(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
request:
        form = getRecords(request.POST)
        # check whether it's valid:
        if form.is_valid():
            fingerprint = form.cleaned_data.get("patient_fingerprint")
            user = User.objects.get(username=request.user)
            clinic_id = user.profile.department_code
            accessRequest = accessControlSC_check(str(fingerprint),
str(clinic_id))
            if accessRequest is True:
                getEHR = patientsSC_ehr_get(int(fingerprint))
                table = ehrTable(getEHR)
                RequestConfig(request).configure(table)
                return render(request, 'dhcare/doctorui-get.html',
{'getEHR':table})
            else:
                return render(request, 'dhcare/doctorui-get.html',
{'error': accessRequest})

        # if a GET (or any other method) we'll create a blank form
        else:
            form = getRecords()

        return render(request, 'dhcare/doctorui-get.html', {"form": form})

@custom_user_passes_test(lambda u: Group.objects.get(name='DHCARE-Doctors')
in u.groups.all())
def doctorui_submit(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the
request:
        form = submitRecords(request.POST)
        # check whether it's valid:
        if form.is_valid():
            fingerprint = form.cleaned_data.get('patient_fingerprint')
            user = User.objects.get(username=request.user)
            clinic_id = user.profile.department_code
            accessRequest = accessControlSC_check(str(fingerprint),
str(clinic_id))
            if accessRequest is True:
                name = form.cleaned_data.get('record_name')
                description = form.cleaned_data.get('record_description')

```

```

        date = str(form.cleaned_data.get('record_date'))
        status = 'Active'
        sha256_name =
str(hashlib.sha256(name.encode()).hexdigest())
        sha256_description =
str(hashlib.sha256(description.encode()).hexdigest())
        H1 = sha256_name + sha256_description
        sha256_date =
str(hashlib.sha256(date.encode()).hexdigest())
        sha256_status =
str(hashlib.sha256(str(status).encode()).hexdigest())
        H2 = sha256_date + sha256_status
        H3 = str(hashlib.sha256(str(H1).encode()).hexdigest()) +
str(hashlib.sha256(str(H2).encode()).hexdigest())
        sha256 = str(hashlib.sha256(H3.encode()).hexdigest())

        createEHR = patientsSC_ehr_post(int(fingerprint), name,
sha256, status, date)
        create_ehr(sha256, description)

        return render(request, 'dhcare/doctorui-submit.html',
{'createEHR': createEHR})
    else:
        return render(request, 'dhcare/doctorui-submit.html',
{'createEHR': accessRequest})

# if a GET (or any other method) we'll create a blank form
else:
    form = submitRecords()

return render(request, 'dhcare/doctorui-submit.html', {"form": form})

```

## Appendix E – Django HTML Templates

about.html

```

{% extends "dhcare/base.html" %}
{% block content %}
<h1>About Page</h1>
    <div class="w3-container">
        <p style="font-family: 'Raleway',sans-serif; font-size: 18px; font-
weight: 500; line-height: 32px; margin: 0 0 24px; text-align: justify;
text-justify: inter-word;"><br>
            DHCARE is Blockchain-Based Healthcare System that utilizes
fingerprints to identify patients and synchronize their
            Electoric Healthcare Records (EHRs) between distributed
healthcare providers.
        <br><br>
        The main features provided by DHCARE are:
        <div style="font-family: 'Raleway',sans-serif; font-size: 18px;
font-weight: 500; line-height: 32px; margin: 0 0 24px; text-align: justify;
text-justify: inter-word;">
            <ul>
                <li>Exchange and synchronization of EHRs between
distributed healthcare providers</li>
                <li>Maintaining EHRs unique patients' identity</li>
                <li>Ensuring recoverable access to patients' EHRs</li>
                <li>Implementing role-based access control to EHRs</li>
                <li>Providing anonymity to patients' EHRs in the cloud
datastores</li>
                <li>Detecting unauthorized modifications to EHRs</li>
                <li>Audit logging any activities on EHRs</li>
            </ul>
        </div>
    </div>

```

```

        </div>
    </p>
</div>
{% endblock content %}

```

### adminui.html

```

{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% load static %}
{% block content %}
<div class="content-section">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Admin Portal</legend>
        </fieldset>
        <br><br>
        <div class="form-group">
            <button class="btn btn-outline-info" formaction="{% url
'adminui_get' %}" type="submit">Get Hospital
            Information
        </button>
            <button class="btn btn-outline-info" formaction="{% url
'adminui_submit' %}" type="submit">Submit Hospital
            Information
        </button>
        </div>
    </form>
</div>
{% endblock content %}

```

### adminui-get.html

```

{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section" xmlns="http://www.w3.org/1999/html">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Hospital
Information</legend>
            {{ form|crispy }}
            {% if provider.provider_name %}
            <div class="text-dark">
                <b> Hospital Name: </b>{{provider.provider_name}}<br><br>
                <b> Hospital Web Address:
</b>{{provider.provider_webAddress}}<br><br>
            </div>
            {% else %}
            <div class="text-danger">
                {{provider}}
            </div>
            {% endif %}
        </fieldset>
        <button class="btn btn-outline-info" type="submit">Get
Info</button>
    </form>

```

```
</div>
{% endblock content %}
```

### adminui-submit.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section" xmlns="http://www.w3.org/1999/html">
  <form method="POST">
    {% csrf_token %}
    <fieldset class="form-group">
      <legend class="border-bottom mb-4">Hospital
Information</legend>
      {{ form|crispy }}
      {% if providerTx %}
      <div class="text-dark">
        <b> {{providerTx}} </b><br><br>
      </div>
      {% else %}
      <div class="text-danger">
        {{providerTx}}
      </div>
      {% endif %}
    </fieldset>
    <button class="btn btn-outline-info" type="submit">Submit
Info</button>
  </form>
</div>
{% endblock content %}
```

### appointments.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section">
  <form method="POST">
    {% csrf_token %}
    <fieldset class="form-group">
      <legend class="border-bottom mb-4">Appointment
Confirmed</legend>
      {{accessControlSC_tx}}
    </fieldset>
  </form>
</div>
{% endblock content %}
```

### base.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>

  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta content="width=device-width, initial-scale=1, shrink-to-fit=no"
name="viewport">
```



```

<!-- Bootstrap CSS -->
<link crossorigin="anonymous"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
"
    <link crossorigin="anonymous"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
"
        integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
rel="stylesheet">

    <link href="{% static 'dhcare/main.css' %}" rel="stylesheet"
type="text/css">

    {% if title %}
    <title>DHCARE - {{ title }}</title>
    {% else %}
    <title>DHCARE</title>
    {% endif %}
</head>
<body>
<header class="site-header">
    <nav class="navbar navbar-expand-md navbar-dark bg-steel fixed-top">
        <div class="container">
            <a class="navbar-brand mr-4" href="{% url 'dhcare-home'
%}">DHCARE</a>
            <button aria-controls="navbarToggle" aria-expanded="false"
aria-label="Toggle navigation"
                class="navbar-toggler" data-target="#navbarToggle"
data-toggle="collapse" type="button">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse" id="navbarToggle">
                <div class="navbar-nav mr-auto">
                    <a class="nav-item nav-link" href="{% url 'dhcare-home'
%}">Home</a>
                    <a class="nav-item nav-link" href="{% url 'dhcare-
about' %}">About</a>
                </div>
                <!-- Navbar Right Side -->
                <div class="navbar-nav">
                    {% if user.is_authenticated %}
                    <a class="nav-item nav-link" href="{% url 'logout'
%}">Logout</a>
                    {% if 'DHCARE-Admins' in user.groups.all.0.name %}
                    <a class="nav-item nav-link" href="{% url 'adminui'
%}">Admin</a>
                    {% endif %}
                    {% if 'DHCARE-Doctors' in user.groups.all.0.name %}
                    <a class="nav-item nav-link" href="{% url 'doctorui'
%}">Doctor</a>
                    {% endif %}
                    {% if 'DHCARE-Reception' in user.groups.all.0.name %}
                    <a class="nav-item nav-link" href="{% url 'receptionui'
%}">Appointments</a>
                    {% endif %}
                    {% else %}
                    <a class="nav-item nav-link" href="{% url 'login'
%}">Login</a>
                    {% endif %}
                </div>
            </div>
        </div>
    </nav>

```

```

</header>
<main class="container" role="main">
  <div class="row">
    <div class="col-md-8">
      {% if messages %}
      {% for message in messages %}
        <div class="alert alert-{{ message.tags }}">
          {{ message }}
        </div>
      {% endfor %}
      {% endif %}
      {% block content %}{% endblock %}
    </div>
  </div>
</main>

<!-- Optional JavaScript -->
<!-- jQuery first, then Popper.js, then Bootstrap JS -->
<script crossorigin="anonymous" integrity="sha384-
KJ3o2DktIkVYIK3UENzmM7KCKRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
<script crossorigin="anonymous" integrity="sha384-
ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min
.js"></script>
<script crossorigin="anonymous" integrity="sha384-
JZR6Spejh4U02d8j0t6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl"
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"><
/script>
</body>
</html>

```

## doctorui.html

```

{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% load static %}
{% block content %}
<div class="content-section">
  <form method="POST">
    {% csrf_token %}
    <fieldset class="form-group">
      <legend class="border-bottom mb-4">Doctor Portal</legend>
    </fieldset>
    <br><br>
    <div class="form-group">
      <button class="btn btn-outline-info" formaction="{% url
'doctorui-get' %}" type="submit">Get Patient Records
    </button>
      <button class="btn btn-outline-info" formaction="{% url
'doctorui-submit' %}" type="submit">Submit Patient
      Records
    </button>
    </div>
  </form>
</div>
{% endblock content %}

```

## doctorui-get.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% load django_tables2 %}
{% block content %}
<div class="content-section" style="width:150%">
  <form method="POST">
    {% csrf_token %}
    <fieldset class="form-group">
      <legend class="border-bottom mb-4">Patient Records</legend>
      {{form|crispy}}
      {% if getEHR %}
        {% render_table getEHR %}
        <div class="form-group">
          </div>
      {% else %}
        {{error}}
      {% endif %}
      <br><button class="btn btn-outline-info" type="submit">Get
Records</button>
    </fieldset>
  </form>
</div>
{% endblock content %}
```

## doctorui-submit.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section">
  <form method="POST">
    {% csrf_token %}
    <fieldset class="form-group">
      <legend class="border-bottom mb-4">Doctor Portal</legend>
      {{form|crispy}}
      {% if createEHR %}
        {{createEHR}}
      {% else %}
        <button class="btn btn-outline-info"
type="submit">Submit</button>
      {% endif %}
    </fieldset>
  </form>
</div>
{% endblock content %}
```

## home.html

```
{% extends "dhcare/base.html" %}
{% load static %}
{% block content %}
  <h1 style="width:150%; color: #666; font-family: 'Consolas',sans-serif;
font-size: 60px; font-weight: 800; line-height: 72px; margin: 0 0 10px;
text-align: center; text-transform: uppercase;">
    DHCARE
  </h1>
  <h2 style="width:150%; color: #987; font-family: 'Calibri',sans-serif;
font-size: 30px; font-weight: 200; line-height: 5px; margin: 0 0 60px;
text-align: center; text-transform: uppercase;">
```

```

        Biometric Distributed Healthcare System on Blockchain
    </h2>
    <br><br><br>
{% endblock content %}

```

### login.html

```

{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Log In</legend>
            {{ form|crispy }}
        </fieldset>
        <div class="form-group">
            <button class="btn btn-outline-info"
type="submit">Login</button>
        </div>
    </form>
</div>
{% endblock content %}

```

### logou.html

```

{% extends "dhcare/base.html" %}
{% block content %}
<h2>You have been logged out</h2>
<div class="border-top pt-3">
    <small class="text-muted">
        <a href="{% url 'login' %}">Log In Again</a>
    </small>
</div>
{% endblock content %}

```

### new-patient.html

```

{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Add Patient</legend>
            {{ form|crispy }}
            {% if confirmation %}
            {{ confirmation }}
            {% else %}
            <button class="btn btn-outline-info" type="submit">Add</button>
            {% endif %}
        </fieldset>
    </form>
</div>
{% endblock content %}

```

## receptionui.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% load static %}
{% block content %}
<div class="content-section">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Reception Portal</legend>
        </fieldset>
        <br><br>
        <div class="form-group">
            <button class="btn btn-outline-info" formaction="{% url 'new-
patient' %}" type="submit">Add New Patient
        </button>
            <button class="btn btn-outline-info" formaction="{% url
'receptionui-get' %}" type="submit">Get Patient
                Appointments
            </button>
            <button class="btn btn-outline-info" formaction="{% url
'receptionui-book' %}" type="submit">Book New
                Appointment
            </button>
        </div>
    </form>
</div>
{% endblock content %}
```

## receptionui-book.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
<div class="content-section" xmlns="http://www.w3.org/1999/html">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Appointment Booking</legend>
            {{ form|crispy }}
        </fieldset>
        <button class="btn btn-outline-info" type="submit">Book</button>
    </form>
</div>
{% endblock content %}
```

## receptionui-get.html

```
{% extends "dhcare/base.html" %}
{% load crispy_forms_tags %}
{% load django_tables2 %}
{% block content %}
<div class="content-section">
    <form method="POST">
        {% csrf_token %}
        <fieldset class="form-group">
            <legend class="border-bottom mb-4">Patient
Appointments</legend>
            {{ form|crispy }}
        </fieldset>
    </form>
</div>
```

```
        {% if table %}
            {% render_table table %}
            <button class="btn btn-outline-info" formaction="{% url
'appointment-confirm' %}" type="submit">Confirm
            </button>
        {% else %}
            <button class="btn btn-outline-info"
type="submit">Get</button>
            {% endif %}
        </fieldset>
    </form>
</div>
{% endblock content %}
```