USING THE BLOCKCHAIN TO IMPROVE INTEROPERABILITY AND PRIVACY IN

ELECTRONIC HEALTH RECORDS

A ███████████████ submitted to the faculty of
San Francisco State University
In partial fulfillment of
the requirements for
the Degree

███████████████████

In

██████████████████████████████████

by

Zun Zhou

San Francisco, California

███████████████ 2023

Certification of Approval

I certify that I have read *Using the Blockchain to Improve Interoperability and Privacy in Electronic Health Records* by Zun Zhou, and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirement for the degree Master of Science in Computer Science: Computer Science at San Francisco State University.

_____

Name of the professor, ███████████

█████████████████████,

Thesis Committee Chair

_____

Name of the professor, ███████████

████████████████

_____

Name of the professor, ███████████

████████████████

_____

Name of the professor, ███████████

████████████████

Abstract

With the healthcare industry transition from paper records to electronic health records (EHR), there has been growing concern on the interoperability and privacy aspects of electronic health records. While hospitals implementing EHR from the same vendor receive good accessibility to the healthcare data, hospitals implementing EHR from different vendors find it difficult to provide adequately care to new patients in emergency situations. Despite legislative effort to protect healthcare data, patients do not have control over who has access to their health data. The focus of this project is to create a consistent EHR framework with patient in charge of the access control by making use of the blockchain. The benefit of the blockchain is that it provides strong availability and integrity. With data integrity in both the access control system and the record itself, patients can take back ownership of their own healthcare data. In this project, we implement a role-based-access-control (RBAC) system onto the consistent and interoperable health records. The prototype developed in this project demonstrates the viability of this approach.

Acknowledgments

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1 Introduction

The current healthcare system relies on electronic health records (EHR) controlled by healthcare providers stored in their respective central databases. 86% of office-based physicians had adopted EHR by 2017 [1]. EHR offer many benefits compared to traditional paper-based records, including improved patient care and increased efficiency. However, there are still difficulties in exchanging data while maintaining privacy and integrity.

1.1 Current Implementations and Challenges

This section will discuss the current implementations of EHR systems on the market and the challenges they face, mainly interoperability, security, and privacy.

1.1.1 Challenges in Interoperability

While EHR systems have data flow to the entities that subscribe to the system, interoperability is poor among entities that subscribe to different EHR systems. Among CancerLinQ supported EHR vendors (e.g., Epic, Cerner, Allscripts), sites implementing the same EHR system tend to be more interoperable [8]. If a patient visits a hospital implementing a different EHR system from their primary provider during an emergency, the lack of information on the patient will result in delayed treatment. National Institute of Health found that interoperable EHR systems improved patient safety and quality of care [4].
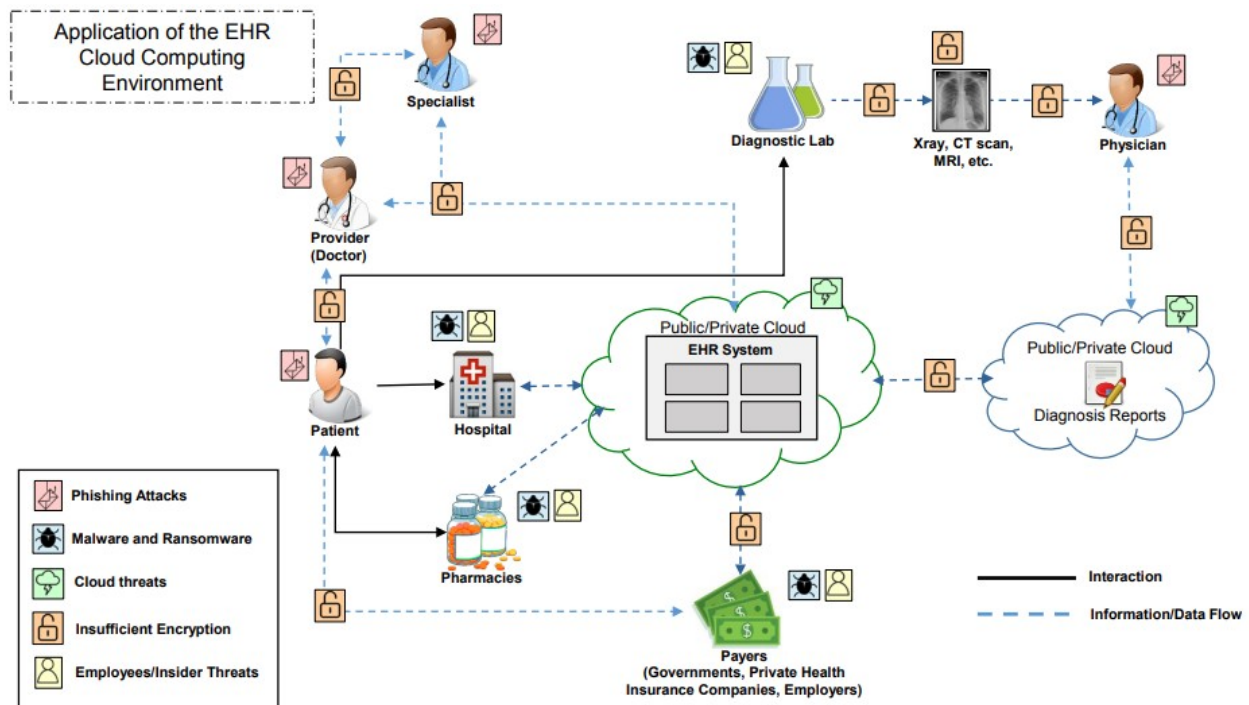
Figure 1: EHR Cloud [1]

1.1.2 Challenges in Security and Privacy

1.1.2.1 Protection Defined in Legislation

Health Insurance Portability and Accountability Act (HIPAA) in the US and General

Data Protection Regulation (GDPR) in the EU are two major pieces of legislation that provide

protection to healthcare data. HIPAA requires covered entities to maintain "reasonable and

appropriate administrative, technical, and physical safeguards" for electronic protected health

information with downstream protections such as confidentiality, security, and breach

notification [10]. The EU counterpart GDPR provides a much higher baseline of protections than

HIPAA. GDPR protects all broadly defined personal data, including health data, both

horizontally and vertically [11]. While HIPAA narrowly applies to covered entities, GDPR

applies to all sectors of the economy. Vertically, GDPR requires organizations to implement appropriate measures to ensure security throughout the lifespan of the data.

1.1.2.2 Challenges Seen in the Market

In a typical EHR system illustrated in Figure 1, patients do no have a integrated view to their records. On the contrary, hospitals, pharmacies, payers, etc. have direct data flow from EHR system. Unpatched systems are prone to malware, ransomware, cloud threats, etc. Take the example of OpenEMR, the platform was found over 20 critical vulnerabilities including cross-site request forgery and unauthenticated information disclosure while being used by over 15 thousands facilities worldwide [9]. Even with the legislation in place, over 25 thousands HIPAA complaints per year are filed since 2018, leading to concerns regarding health information privacy and patients' willingness to share medical data [5][6][7]. Extending to third-party entities, major insurance companies suffered data breaches in 2015 that affected approximately 100 million records [18]. It would be a key advantage to put trust back into EHR if patients can take ownership of their records.

1.1.2.3 Challenges from the Changing Landscape

New technologies and business models makes enforcement of these rules harder and harder [3][20]. The rise of wearable and internet of things is posing new challenges for protecting health data. HIPAA requires healthcare data to be available upon patients' request. Wearable often rely on user health data to perform accurate analytic. Once patients request their health data from the healthcare providers, the data is no longer protected under HIPAA. These devices also generate tremendous amount of data that is not covered by HIPAA. Therefore, they are often not held to the same security and privacy standard [2].

1.2 Our Solution
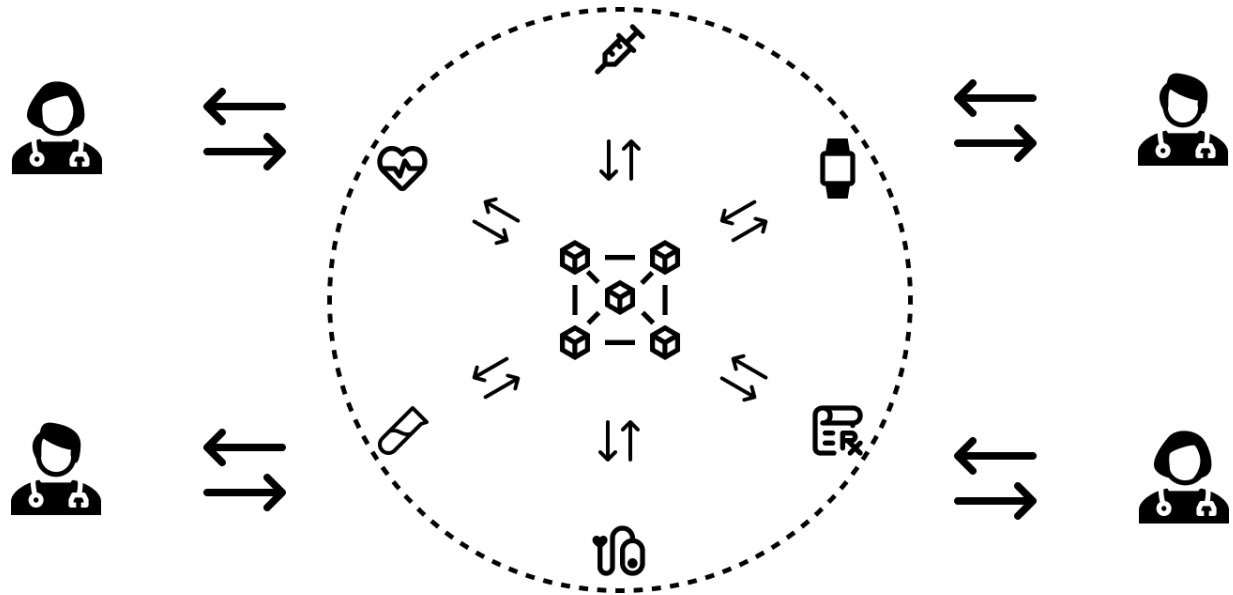
Figure 2: EHR Ecosystem

In this project, the goal is to improve interoperability while preserving security and privacy of EHR using the blockchain technology. Being a general purpose blockchain, Ethereum allows developers to build applications on top of the health records, forming an ecosystem similar to those with public API, illustrated in Figure 2. Blockchain wallet provides a unique identifier for the user. To preserve security and privacy, in our EHR system, Patients are the owner of their records. We also implement a role-based-access-control on the health records, where patients would be able to grant doctors and applications access. Being a decentralized platform, blockchain provides strong data integrity and availability, protecting against data corruption, server outage and ransomware attack found in traditional infrastructure.

When needed, doctors with access can interact with patient records through applications. To demonstrate, we build a standalone medication reminder application to take advantage of the

EHR system. We have seen similar application before such as Medisafe [12], and Apple's Medications in the Health app [13]. Our solution connect doctors and patients through the ecosystem, allowing doctor to see patients' latest records and prescribe medications directly.

Chapter 2 Background and Related Work

This chapter introduces the technologies used in this project. First we explain the security concepts to preserve privacy to EHR, then we will introduce blockchain technologies, followed by ways to reduce cost and additional security framework.

2.1 Security Frameworks

This section will introduce some of the security concepts implemented in this project. We will mainly look at authentication and authorization. Authentication is the process of verifying that an entity is whom it claims to be, often as a prerequisite to allowing access to resources in an information system Error: Reference source not found. Authorization, also known as Access Control, is the process of verifying that a requested action or service is approved for a specific entity Error: Reference source not found.

2.1.1 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol designed to provide secure communication over a computer network Error: Reference source not found. Besides encrypting the messages sent over the network, TLS authenticates the entities exchanging information and ensures the integrity of the messages. In this project, we will use TLS to encrypt the HTTP communication between client and server. To establish a TLS session, client and server will perform a TLS handshake illustrated in Figure 3. The client first send a "client hello" message to the server. The server then encrypts a "server hello" message with server's private key, and send it to the client along with server's certificate that includes server's public key. The client can verify the certificate with a Certificate Authority (CA) and decrypt the server message with

server's public key confirming server's identity. The client performs the key exchange using the "server hello" message then sends the cipher spec to the server. The server receives the cipher spec, the TLS session is established. Then both sides will encrypt messages using the cipher spec onward.
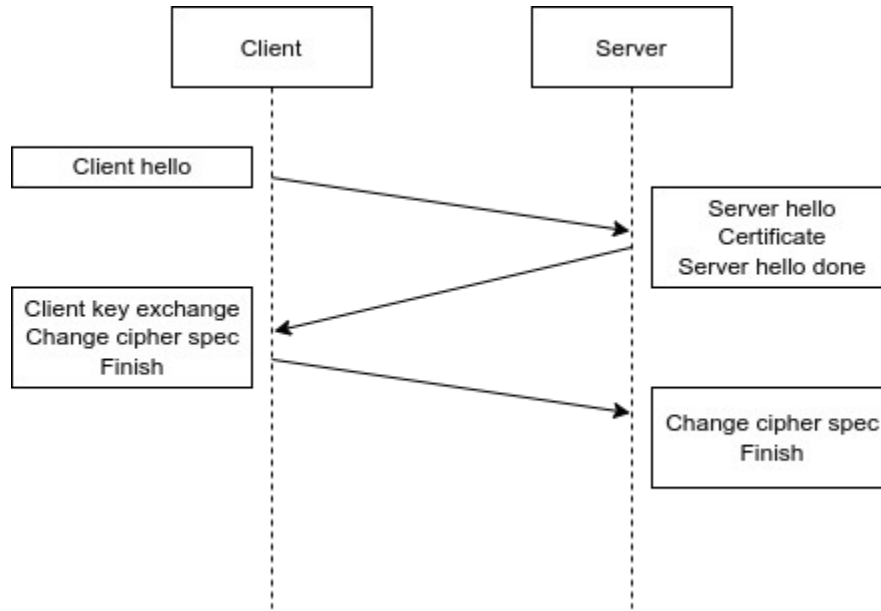
Figure 3: TLS Handshake

2.1.2 Access Control

In this project, we are implementing two access control concepts, Ownership-Based Access Control (OBAC) and Role-Based Access Control (RBAC). OBAC is the concept of a single owner with elevated privilege in a system [1]. When there are different levels of privilege, RBAC is the alternative system to implement, in which the actions users allowed to performed are defined by the roles they acquired [41][42].

2.2 Blockchain

Blockchain is a digital ledger on a distributed network [24]. Each block on the chain contains the hash of the last block, the hash of this block, and a list of transactions illustrated in Figure 4. Every computer in the network must agree upon the creation of new blocks, also known as consensus. In a most popular consensus mechanism, Proof-of-Work (PoW), reaching the consensus requires a process called mining, which is the process of finding a nonce value that makes the block hash meet a certain requirement [32]. Hash function is a type of one-way

function that maps data of arbitrary size to unique fixed-size values. Due to hash functions' one-way nature, a hash value can not be easily reverted to the original input. Therefore, the nonce value has to be brute forced. The first miner who finds a nonce value that can generate a valid block hash confirms all the transactions on the block and broadcast to the rest of the network. A new block is then added to the blockchain and the miner is rewarded with tokens.
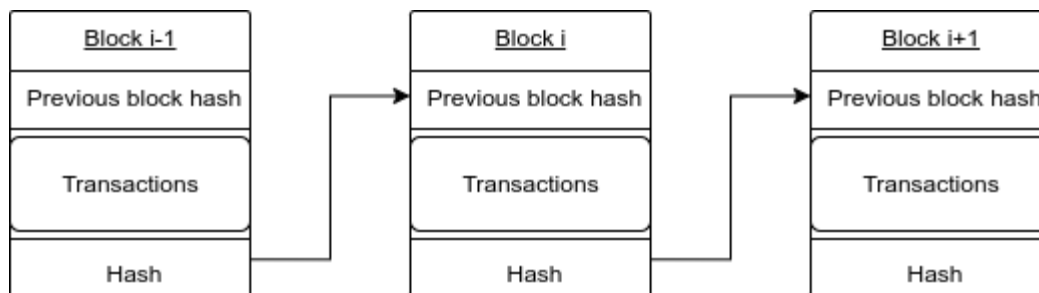
Figure 4: Blockchain Structure

If an attacker wants to make changes to an existing block, they would have to re-mine the block and every following blocks. With the whole network racing to mine new blocks, it is nearly impossible to tamper with existing block. With the distributed network, blockchain has the following key karacteristics:

- Decentralization. Blockchain technology is not controlled by any single entity, which means it can not be censored and would not have down time.

- Immutable. Transactions on the blockchain are verified through the consensus mechanism, and it is nearly impossible to be altered or deleted.

- Transparent. All transactions on the blockchain can be queried and audited.

2.3 Ethereum

Ethereum is a blockchain embedded with a computer, allowing applications and organizations to be built in a decentralized, permissionless, censorship-resistant way [23]. Ethereum Virtual Machine (EVM) is the single, canonical computer whose state is distributed across the entire network. This computer provides the environment for smart contracts to execute computation. Users can request for computation through transactions; all transactions and the EVM's state are validated and stored on the blockchain, which is agreed upon by the entire network. Every transaction on the Ethereum blockchain includes a gas fee. Gas is the unit that measure the computational resource required to execute transactions.

2.3.1 Proof-of-Stake

Ethereum switched to a more secure, less energy-intensive Proof-of-Stake (PoS) consensus mechanism[31]. In a PoS consensus, validators create new blocks and receive coins in return. Users can stake their coins to participate as a validator node. The amount of stake affects the chances of a node being selected. To prevent the system being biased towards the wealthiest node, additional mechanisms are added to the selection process. For example, some PoS system uses a Coin Age Selection mechanism where it takes in the factor of how long the coins of a node has been staked. The system will lean towards selecting the nodes with coins that are staked for longer. Once the node is selected the age of the coins will be reset, so there will be a cool down period before the node is selected again.

Once a validator is selected, they receive new blocks from peers from the network. The validator then finds the block that contains the valid transactions and block signature, and sends a vote in favor of that block to the rest of the network. To prevent a validator from acting

maliciously, the stake to validate a block must be higher than the transaction total in the block. If the validator approved fraudulent transaction, their stake along with the rewards will be burned.

Resistant to 51% Attack

51% attack is an attack on a blockchain network by a group of miners who own more than 50% of the computing power. Since blockchain networks run on a majority rules democracy, owning over 50% of the network allows the attackers to reverse transactions, block transactions, changing the transaction history on the blockchain. Figure 5 shows that at the time of writing, if two of the biggest mining pools merged, they would have control to the bitcoin network. In a PoS consensus, an attacker would need to own over 50% of the circulating supply
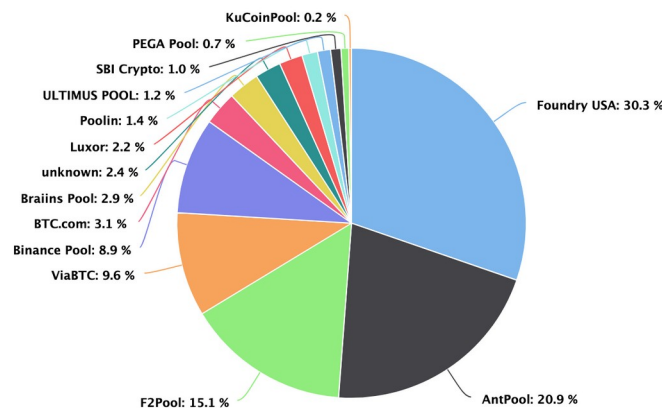
on a PoS network would be infeasible.

Decentralization

In a PoW consensus where miners with more computing power are more likely to win the race to create new blocks, participating in the mining process can be infeasible for most users [31]. The cost of mining also increases as new blocks are added to the blockchain in PoW.

Therefore, PoW consensus tends towards centralization. PoS consensus does not require large amount of computing power to create new blocks, which allows more users to setup validator nodes without purchasing expensive hardware. Therefore, a PoS system tends towards decentralization.

2.3.2 Smart Contract

A smart contract is a piece of code that is deployed into EVM state. Any user can request to execute the code in smart contract by making transaction request [33]. Other users on the network can then verify, validate and execute the computation. These transaction requests include a gas fee just like other transactions. To avoid infinite loop and other malicious operations, a gas limit is posed on every transaction. In the following section, we will discuss a method to reduce gas fee for certain operation

2.3.3 Minimal Proxy Contract
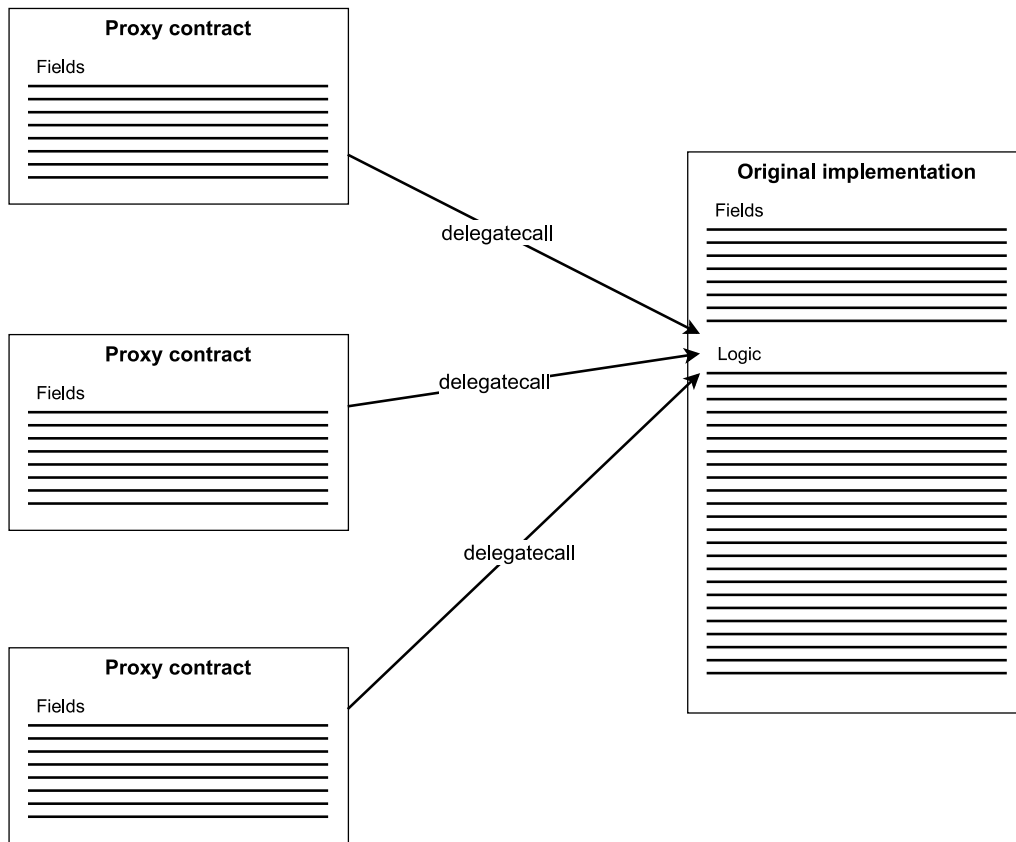
Figure 6: Minimal Proxy Contract

Minimal Proxy Contract, also known as Clones, is proposed in Ethereum Improvement Proposals EIP-1167 which allows developers to simply and cheaply clone contract functionality in an immutable way [25][34]. Deploying new contracts to the blockchain is an expensive operation. Certain contracts such as user profile, game session are required to be deployed multiple times in the lifespan of an application, which causes the application to be expensive to use.
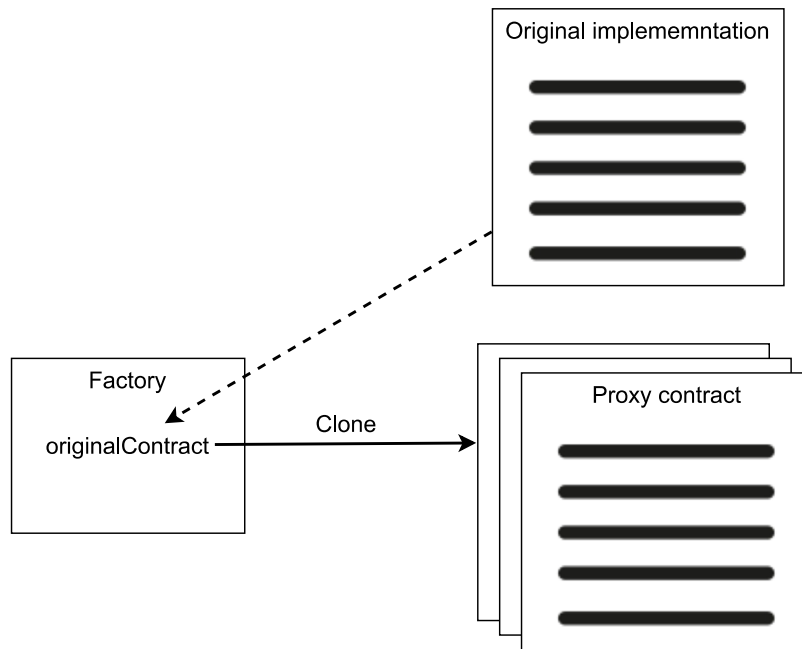
Figure 7: Minimal Proxy Contract Deployment Workflow

Figure 6 shows that Minimal Proxy Contract solves this problem using delegatecall, which allows a contract to execute functions of another contact in the context of the original contract [39]. Figure 7 shows that deploying minimal proxy contract workflow is similar to a factory pattern where developers first deploy a factory contract that contains the original implementation contract that they want to deploy multiple times. To create a copy of the implementation contract, the factory contract create a proxy contract to the implementation contract. This proxy contract only contains the fields of the implementation contract but not the logic. When a function of the proxy contract is called, the proxy contract will make a delegate call to the implementation contract. Since the context of the delegate call is in the proxy contract, the data will be modified in the proxy contract if needed, and the implementation contract will remain unchanged. Figure 8 illustrates using two common Ethereum examples that the initial

deployment of the factory cost slightly more gas for developers with clone, but it is much

cheaper in the long run for users.
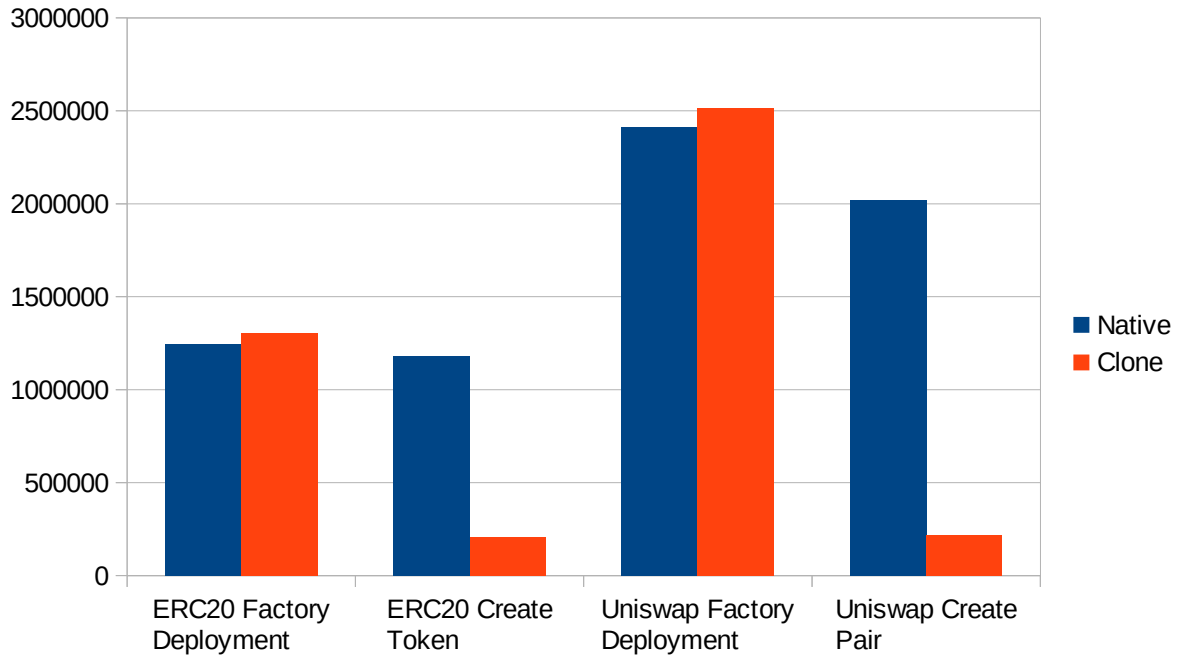
Figure 8: Minimal Proxy Contract Deployment Cost [35]

2.3.4 Sign-in with Ethereum

Sign-in with Ethereum is proposed in Ethereum Improvement Proposal EIP-4361, which

allows users to authenticate with their Ethereum wallet for off-chain services. Signing into off-

chain services nowadays often turns to large centralized identity providers (IdPs) through OAuth

or SAML. While this method eliminates the need to create new usernames and passwords, users'

identities are controlled by the IdPs, which leads to more aggregation of user data. Since

Ethereum already provided users an built-in identifier, Ethereum wallet, sign-in with Ethereum is

a self-custodial alternative to centralized IdPs.
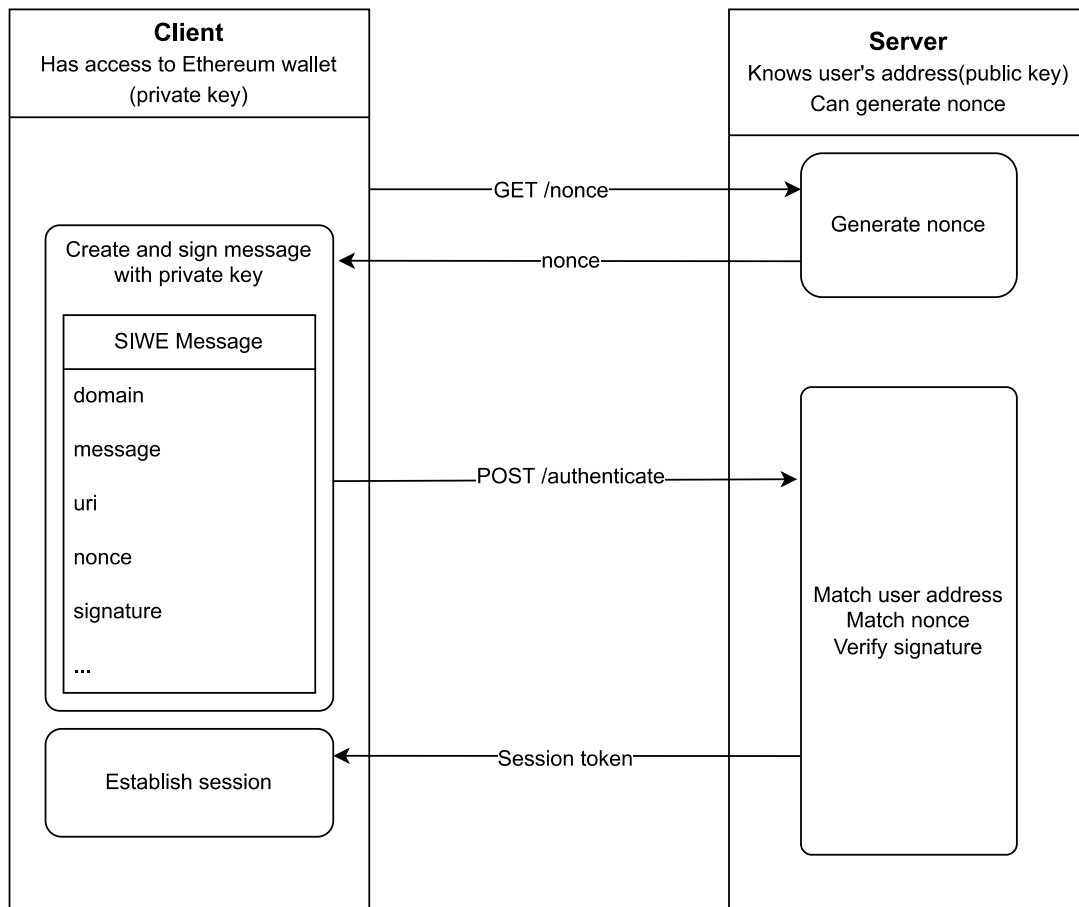
Figure 9: Sign-in with Ethereum

To initiate Sign-in with Ethereum, the Ethereum wallet first provides the user a structured message which contains an address, domain requesting the signing, version of the message, chain-id, uri for scoping, nonce as security mechanism, and issued-at time stamp. The user then sign the message with their private key and send it to the off-chain service. Finally, the off-chain service can validate the user identity with the signature in the message. Figure 9 illustrates the sign-in with Ethereum workflow.

Chapter 3 Design and Architecture

This chapter will illustrate the design and architecture of this project.

3.1 User Stories

3.1.1 Scenario 1: patients share medical information to authorized personnel

Alice, a college student who is recently diagnosed with a rare allergy. Soon after the diagnosis, Alice starts seeing advertisement about a potential cure for her allergy while she browse the internet. It turns out that the hospital Alice visited failed to remove personal identifiable information when sharing patient files with pharmaceutical companies for research. Alice wants a medical record that she can choose who has the access to.

3.1.2 Scenario 2: doctor prescribe medication to returning patients

Bob, a doctor, is in the process of prescribing a new medication for one of his returning patients. Little did he know, the patient forgot to mentioned that he had an emergency visit at a different hospital during a business trip, during which the patient was put on a medication that would conflict with the medication Bob is about to prescribe. Bob wants a health record that allow him to see whether his patients are on medications that are prescribed elsewhere.

3.2 System Requirements

System requirements are extracted from the user stories in previous section and summarized below. To better understand the design decisions made on the project, we need to distinguish the two components in the project, "Manager" and "Application". Manager represents the EHR system that stores doctor and patient records independent from Application. Application relies on manager to retrieve doctor and patient records while doubling as a web

interface for Manager in our case. We wanted to demonstrate improved interoperability by decoupling Manager from Application, so that other Applications can take the address of the Manager and gain access to the doctor and patient records through the same authorization process.

- The user shall be able to create a patient.

- The user shall be able to create a doctor.

- The user, as a doctor, shall be able to request access to patients.

- The user, as a patient, shall be able to update their information.

- The user, as a patient, shall be able to grant access to applications.

- The user, as a patient, shall be able to accept access request from doctors.

- The user, as a doctor, shall be able to prescribe treatment to patients.

- The user, as patient, shall be able to report taking medication for treatments.

- The user, as doctor, shall be able to send notification to remind patients to take their medications.

- The user shall be able to receives reminder notification for off-track treatments

3.3 Architecture

3.3.1 Overview
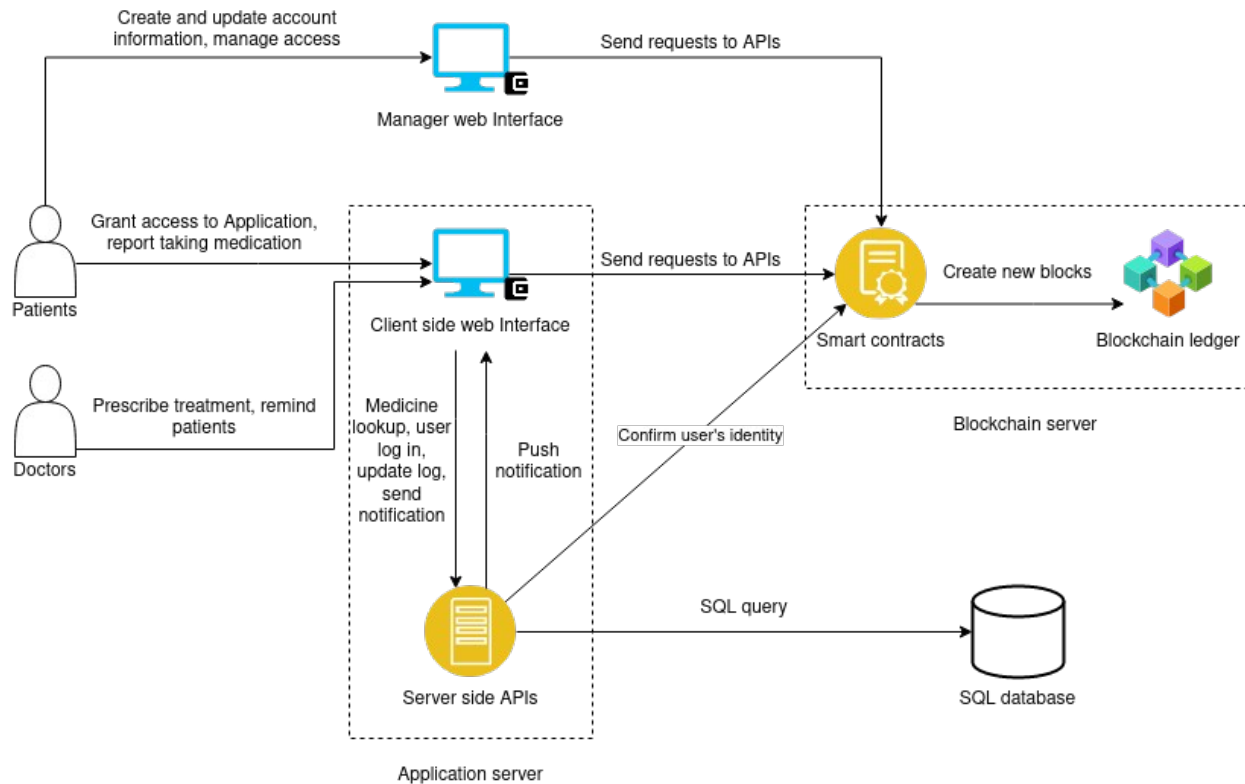
Figure 10: Architecture Overview

Figure 10 shows the key components of our application including Manager web interface, Application server, SQL database server, and Blockchain server. Users interact with the blockchain and server side APIs through client side web interface from Application server and Manager web interface. As mentioned in earlier chapter, blockchain provides integrity and accessibility but it is inefficient. Traditional SQL database has better scalability, but it is prone to data corruption and outages. The hybrid setup for the Application here allows us to take advantage of the immutability of blockchain and efficiency of SQL database. Data that require integrity such as patient records and access control list are stored on the blockchain. Other data such as medication lookup and patient report logs are stored in SQL database. Here we are using the Ethereum implementation of blockchain server with smart contracts providing the logic and

blockchain ledger as storage. For patient reporting their progress, only the patients' wallet address and the treatments' contract address are stored to preserve privacy. Server side APIs checks the access control on the treatment contract before making query to the SQL database.

3.3.2 Sequence Diagrams

There are two major workflows in this application. The first one is the patient related workflow, including creating their profile, granting access to doctor, reporting taking the medication for their treatments. The second one is the doctor related workflow, including requesting access to patients' profile, prescribing treatments to patients.

3.3.2.1 Patient Related Workflow
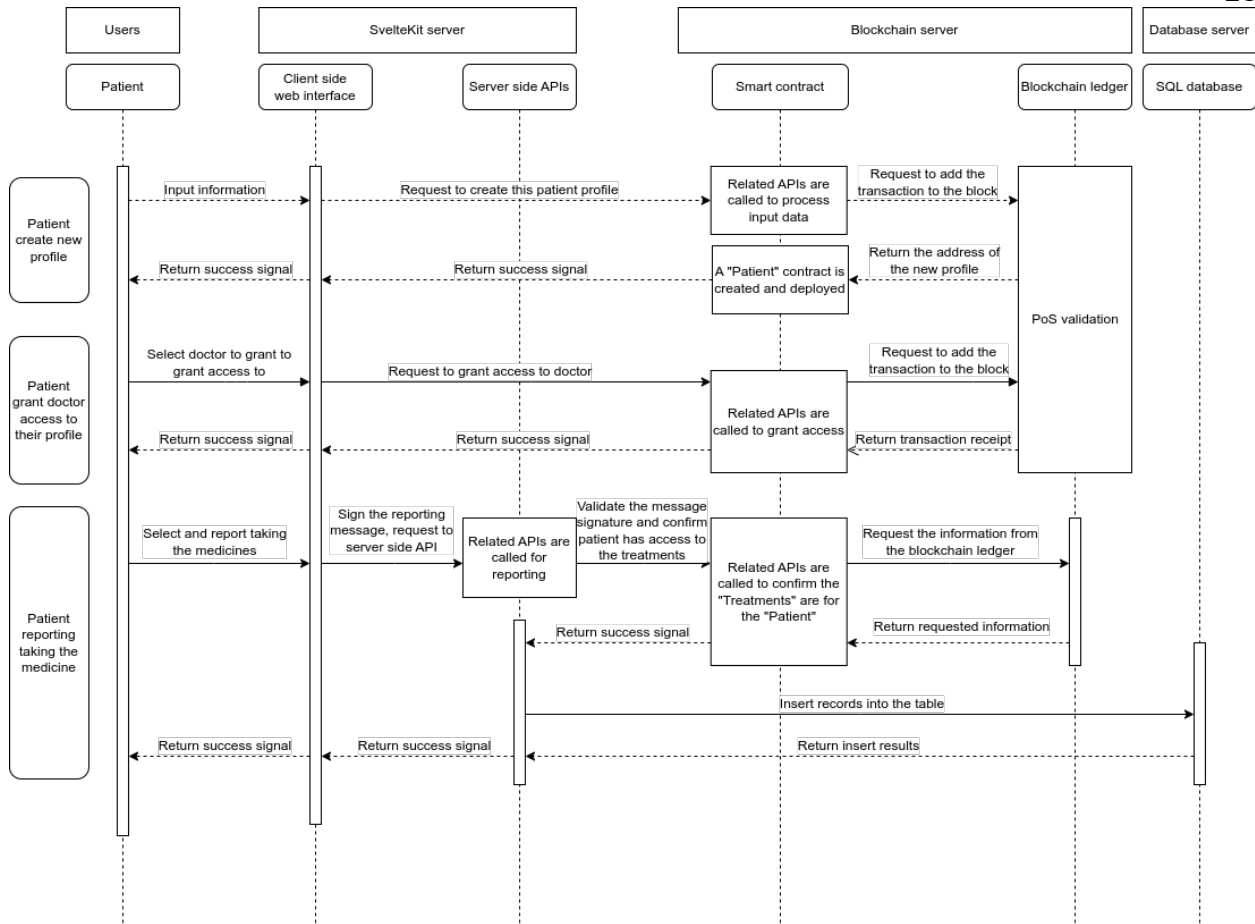
Figure 11: *Patient* Related Sequence Workflow

As illustrated in Figure 11, the user can input their information to create a patient

contract. Once the patient contract is created, the user can log on to the application through the

Sign-in with Ethereum process. Although the user client already established session with the

server after login, we opted to implement the same Sign-in with Ethereum process for reporting

taking medication for a specific treatment to increase security. We will introduce details of

related APIs in Chapter 4.

3.3.2.2 Doctor Related Workflow

Figure 12 illustrates doctor's workflow in prescribing a new treatment to a patient. The doctor would first check whether they have the access to prescribing treatment to the patient. If the doctor has the access, they then query the medication information from the off-chain SQL database. The doctor fills out the necessary information of the treatment and prescribe the treatment to the patient. If the doctor does not have the access, they then have to request the access before being able to continue.
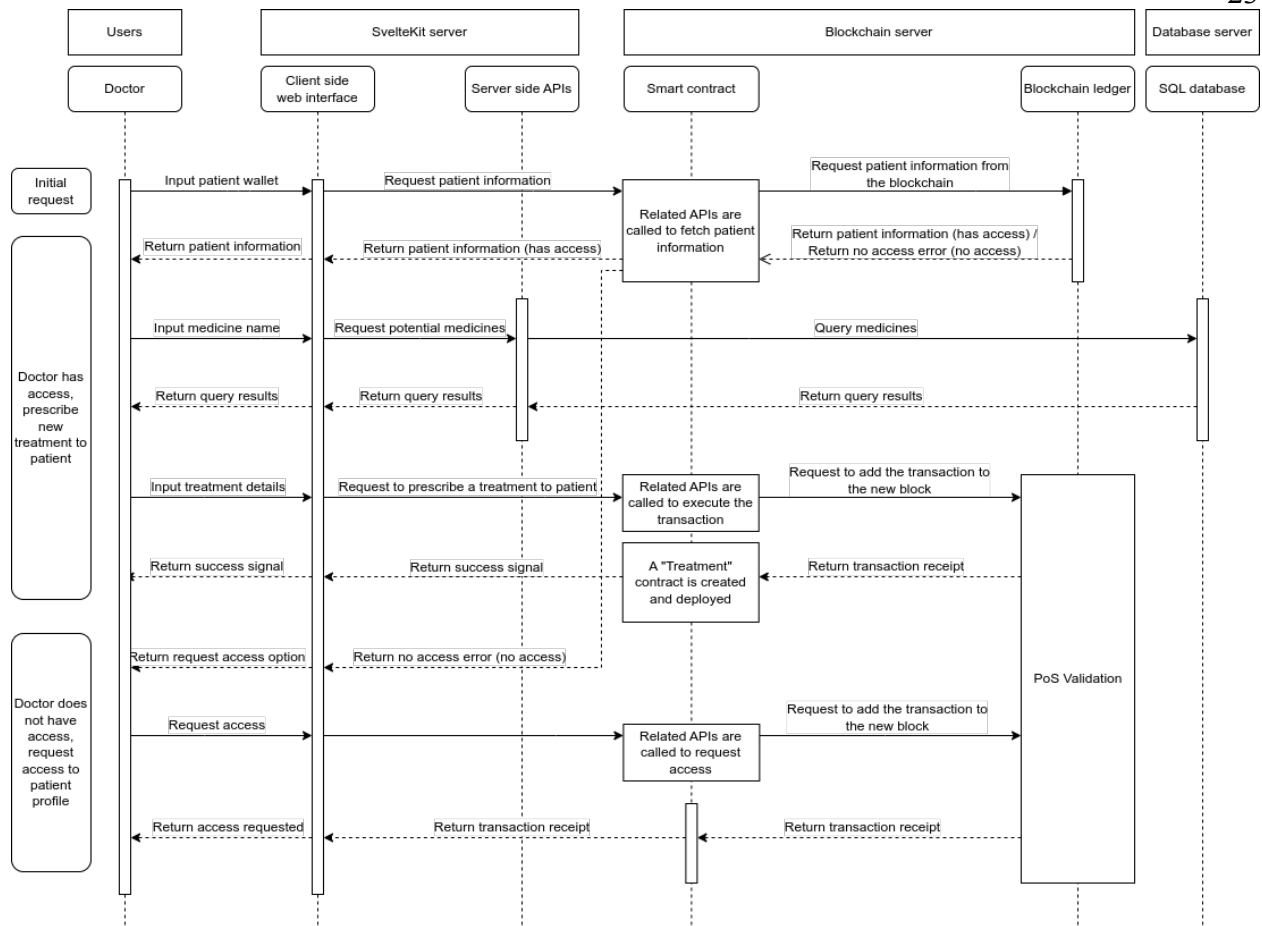
Figure 12: Doctor Related Sequence Workflow

Chapter 4 Implementation

4.1 UML Diagram

Smart contracts are similar to classes in object-oriented programming. We can model them with UML diagrams shown in Figure 13.
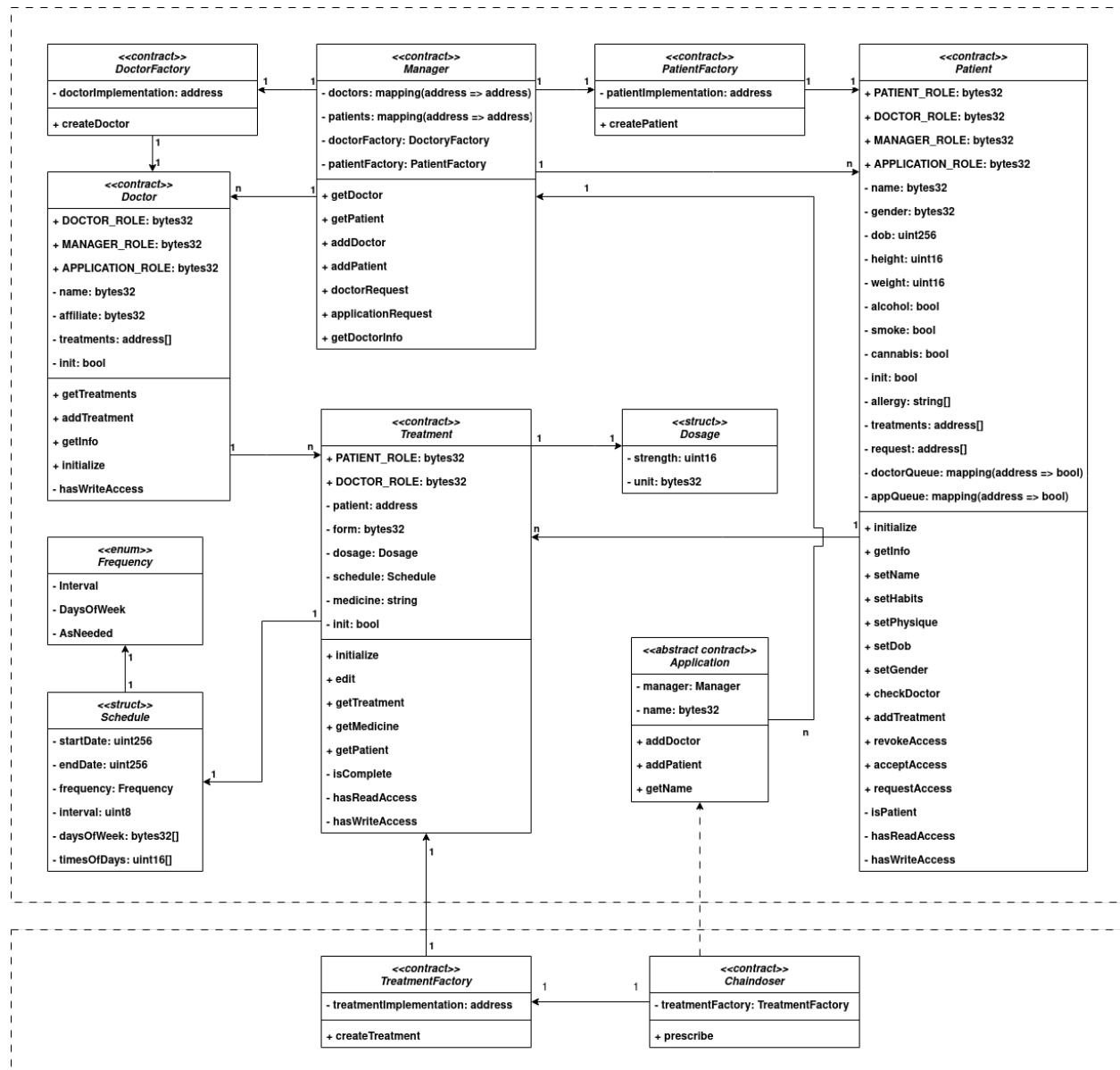
Figure 13: Smart Contracts *UML* Diagram

Before diving into the specific smart contracts, four helper structs are used only in memory to reduce the number of variables for some functions summarized in Table 1. Additionally, two contracts from OpenZeppelin are used to implementing access control summarized in Table 2.

| Structs	Fields	Notes
Bytes32Pair	value1: bytes32 value2: bytes32	A struct to bundle name and gender together when deploying Patient contract
BoolTriple	value1: bool value2: bool value3: bool	A struct to bundle alcohol, smoke, and cannabis together when deploying Patient contract
SmallUintPair	value1: uint16 value2: uint16	A struct to bundle height and weight together when deploying Patient contract
FromAppOption	fromApp: bool addr: address	A struct to pass in as option for Manager to distinguish whether the Patient or Doctor contract is deployed through an application to grant proper access to the message sender.

Table 1: Helper Structs

Contracts	Children	Notes
Ownable	DoctorFactory PatientFactory TreatmentFactory	Ownable implements the concept of ownership mentioned in chapter 2.1.2, which ensure certain functions be only called by the owner of the contract. In this instance, DoctorFactory and PatientFactory are owned by Manager, TreatmentFactory is owned by Application
AccessControl	Doctor Patient Treatment	AccessControl implements RBAC mentioned in chapter 2.1.2, allowing different users/contracts to have access to functions based on their roles.

Table 2: Access Control Contracts

The top dashed-rectangle represent the EHR ecosystem, and the bottom dashed-rectangle is the example medication reminder application that is accessing patient records in this EHR system. Manager contract is the EHR system in this project. This contract is deployed prior to Application contract, then the Application contract can be deployed with the Manager contract

address. Manager contract implement AccessControl that includes ADMINISTRATOR_ROLE

required to add new doctors to the EHR system. Manager contract contains two mappings,

doctors and patients. These two mappings map the account address of the user to their doctor or

patient contract address. Manager contract also contains the DoctorFactory and PatientFactory

for creating new doctor and patient contracts. Manager contract's functions are summarized in

Table 3

Functions	Parameters		Notes
addDoctor	Input(s)		Calls to DoctorFactory to deploy a new Doctor contract and save it in the doctors mapping. The fromApp_ parameter indicates whether this is called from an application. If true, the application would be granted the APPLICATION_ROLE. This function could only be called by users with ADMINISTRATOR_ROLE.
	owner_: address name_: bytes32 affiliate_: bytes32 fromApp_: bool		
	Return		
	none		
addPatient	Input(s)		Calls to PatientFactory to deploy a new Patient Contract and save it in the patients mapping. Some of the parameters here are structs to reduce the numbers of variables passed in, summarized in Table 1. Patients would call this function with their own account, and their account address would be granted the PATIENT_ROLE.
	nameGender_: Bytes32Pair dob_: uint256 physique_: SmallUintPair, allergy_: string[] habits_: BoolTriple option_: FromAppOption		
	Return		
	none		
getDoctor	Input(s)		Retrieves Doctor contract address by doctor's account address
	owner: address		
	Return		
	doctor: address		
getPatient	Input(s)		Retrieves Patient contract address by patient's account address
	owner: address		
	Return		

	patient: address	
requestAccess	Input(s)	Request a DOCTOR_ROLE to a Patient contract
	owner_: address	
	Return	
	none	

Table 3: Manager contract functions

DoctorFactory contains the original implementation of the Doctor contract.

DoctorFactory implements Ownable and is owned by Manager contract to cheaply deploy new

Doctor contracts. DoctorFactory contract's functions are summarized in Table 4

Functions	Parameters	Notes
createDoctor	Input(s)	Clones and initialize a new Doctor contract from the implementation with the doctor information passed in from Manager. This function can only called by the owner of this DoctorFactory, which is the Manager contract.
	owner_: address name_: bytes32 affiliate_: bytes32 option_: FromAppOption	
	Modifier	
	onlyOwner	
	Return	
	doctor: address	

Table 4: DoctorFactory contract functions

Doctor contract contains the doctor's name, affiliate and an array of Treatment contracts'

address. Doctor contract implement AccessControl, and the role hierarchy is illustrated in

Figure14. DOCTOR_ROLE is the role admin for MANAGER_ROLE and

APPLICATION_ROLE. Although all three roles have read and write access, only

DOCTOR_ROLE can grant MANAGER_ROLE and APPLICATION_ROLE to other addresses.

Doctor contracts' function are summarized in Table 5
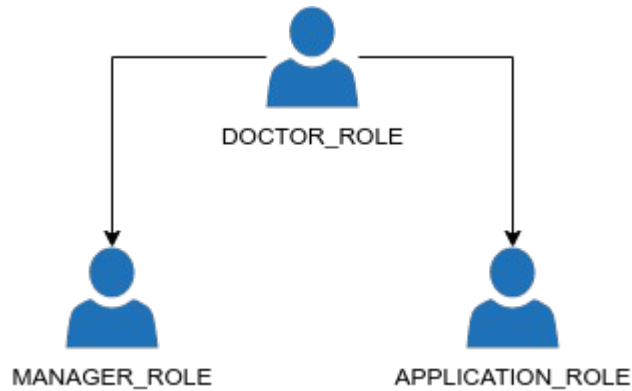
Figure 14: Doctor Contract Role Hierarchy

Functions	Parameters		Notes
initialize	Input(s)		Initializes the contract after it is cloned from the implementation with the necessary information and set up the proper roles. This function would only be called once
	owner_: address app_: address manager_: address name_: bytes32 affiliate_: bytes32		
	Return		
	none		
getInfo	Input(s)		Retrieves the basic information of the doctor
	none		
	Return		
	name_: bytes32 affiliate_: bytes32		
addTreatment	Input(s)		Pushes a new treatment address to the treatments array. Only accounts/contracts with write privilege can call this function
	treatment_: address		
	Modifier		
	canWrite		
	Return		
	none		

getTreatments	Input(s)	Retrieves the treatments array. Only accounts/contracts with the DOCTOR_ROLE can call this function
	none	
	Modifier	
	onlyRole(DOCTOR_ROLE)	
	Return	
	treatments: address[]	
hasWriteAccess	Input(s)	A private helper function for the canWrite modifier
	none	
	Return	
	none	

Table 5: Doctor contract functions

PatientFactory contains the original implementation of the Patient contract.

PatientFactory implements Ownable and is owned by Manager contract to cheaply deploy new

Patient contracts. PatientFactory contract's functions are summarized in Table 6.

Functions	Parameters	Notes
createPatient	Input(s)	Clones and initialize a new Patient contract with the patient information passed in from Manager. This function can only called by the owner of this PatientFactory, which is the Manager contract.
	owner_: address nameGender_: Bytes32Pair dob_: uint256 physique_: SmallUintPair, allergy_: string[] habits_: BoolTriple option_: FromAppOption	
	Modifier	
	onlyOwner	
	Return	
	patient: address	

Table 6: PatientFactory contract functions

Patient contract contains some basic information about the patient as well as an array to

hold treatment addresses. Additionally, there is an array of address for the request queue, as well

as a doctor queue and an application queue, both mappings of address to bool to determine which role the address is seeking. Patient contract implements AccessControl, and the role hierarchy is illustrated in Figure 15. PATIENT_ROLE is the role admin for all DOCTOR_ROLE, MANAGER_ROLE, and APPLICATION_ROLE. Although all four roles have read and write access, only DOCTOR_ROLE can prescribe new treatment to patient, more on that in next paragraph. Patient contract's functions are summarized in Table 7
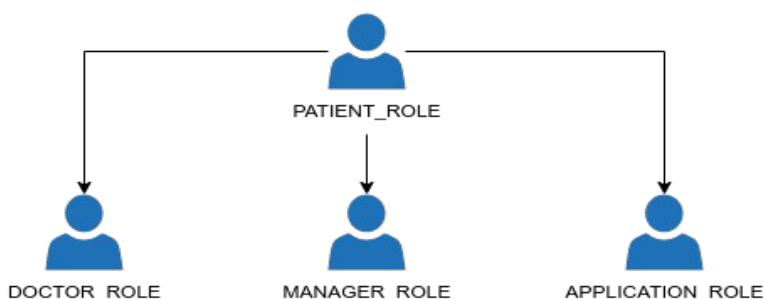
Figure 15: Patient Contract Role Hierarchy

Functions	Parameters	Notes
initialize	Input(s)	Initializes the contract after it is cloned from the implementation with the necessary information and set up the proper roles. This function would only be called once
	manager_: address owner_: address app_: address nameGender_: Bytes32Pair dob_: uint256 physique_: SmallUintPair, allergy_: string[] habits_: BoolTriple	
	Return	
	none	

getInfo	Input(s)	Retrieves patient's information, requires read access
	none	
	Modifier	
	canRead	
	Return	
	name: bytes32 gender: bytes32 dob: uint256 height: uint16 weight: uint16 allergy: string[] alcohol: bool smoke: bool cannabis: bool treatments: address[] requests: address[]	
setName	Input(s)	Updates patient's name. Only patient themselves can call this function
	name_: bytes32	
	Modifier	
	onlyPatient	
	Return	
	none	
setGender	Input(s)	Update patient's gender. Only patient themselves can call this function
	gender_: bytes32	
	Modifier	
	onlyPatient	
	Return	
	none	
setDob	Input(s)	Update patient's date of birth. Only patient themselves can call this function
	dob_: uint256	
	Modifier	
	onlyPatient	
	Return	

	none	
setPhysique	Input(s)	Update patient's height and weight, requires write access
	physique_: SmallUintPair	
	Modifier	
	canWrite	
	Return	
	none	
setHabits	Input(s)	Update patient's habits (alcohol, smoke, cannabis), require write access
	habits_: BoolTriple	
	Modifier	
	canWrite	
	Return	
	none	
checkDoctor	Input(s)	Checks whether the user has the DOCTOR_ROLE. This should be called by applications before writing to the Patient contract on the doctor's behalf
	user_: address	
	Modifier	
	canWrite	
	Return	
	bool	
addTreatment	Input(s)	Add new treatment to the treatments array, requires write access
	treatment_: address	
	Modifier	
	canWrite	
	Return	
	none	
requestAccess	Input(s)	Request access to the Patient contract.
	requester_: address type_: RequestType	
	Return	
	none	

acceptRequest	Input(s)	Grant the requested role to the requester. Only patient themselves can call this function
	requester_: address type_: RequestType	
	Modifier	
	onlyPatient	
	Return	
	none	
getRequestQueue	Input(s)	Retrieves the request queue. Only patient themselves can call this function
	none	
	Modifier	
	onlyPatient	
	Return	
	queue: address[]	
revokeAccess	Input(s)	Strips the role of the requester_. Only patient themselves can call this function
	requester_: address	
	Modifier	
	onlyPatient	
	Return	
	none	
hasReadAccess	Input(s)	A private helper function for the canRead modifier
	none	
	Return	
	none	
hasWriteAccess	Input(s)	A private helper function for the canWrite modifier
	none	
	Return	
	none	
isPatient	Input(s)	A private helper function for the onlyPatient modifier
	none	
	Return	
	none	

Table 7: Patient contract functions

Application is an abstract contract that other application can inherit from. Application contract's constructor takes in a Manager contract address and a name, since that is the contract Application would use for registering new patients or doctors with APPLICATION_ROLE already granted. Application contract's functions are summarized in Table 8

Functions	Parameters	Notes
constructor	Input(s)	This takes in the Manager contract address and a name required to deploy this contract
	manager_: address name_: bytes32	
	Return	
	none	
addDoctor	Input(s)	Calls the Manager contract to deploy a new Doctor contract which automatically grants the APPLICATION_ROLE to this contract
	owner_: address name_: bytes32 affiliate_: bytes32	
	Return	
	none	
addPatient	Input(s)	Calls the Manager contract to deploy a new Patient contract which automatically grants the APPLICATION_ROLE to this contract
	nameGender_: Bytes32Pair dob_: uint256 physique_: SmallUintPair allergy_: string[] habits_: BoolTriple	
	Return	
	none	

Table 8: Application contract functions

Chaindoser is the contract for the medication reminder application that we are building, this contract inherit the Application contract. Chaindoser contract also contains a

TreatmentFactory to cheaply deploy Treatment contracts. Chaindoser contract's functions are

summarized in Table 9.

Functions	Parameters	Notes
prescribe	Input(s)	Checks the user's privilege then prescribes a new treatment for the patient
	patient_: address medicine_: string form_: bytes32 dosage_: Dosage schedule_: Schedule	
	Return	
	none	

Table 9: Chaindoser contract functions

TreatmentFactory contains the original implementation of the Treatment contract.

TreatmentFactory implements Ownable and is owned by Application contract to cheaply deploy

new Treatment contracts. TreatmentFactory contract's functions are summarized in Table 10.

Functions	Parameters	Notes
createTreatment	Input(s)	Clones and initialize a new Treatment contract with the treatment information passed in from Application. This function can only called by the owner of this TreatmentFactory, which is the Application contract
	doctor_: address patient_: address medicine_: string form_: bytes32 dosage_: Dosage schedule_: Schedule	
	Modifier	
	onlyOwner	
	Return	
	treatment: address	

Table 10: TreatmentFactory contract functions

Treatment contract contains the doctor, patient address, medicine, form, dosage and

schedule. Treatment contract implements AccessControl with the role hierarchy illustrated in

Figure 16. DOCTOR_ROLE is the role admin for PATIENT_ROLE. DOCTOR_ROLE has read

and write access, while PATIENT_ROLE only has read access. Treatment contract's functions
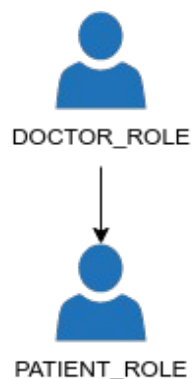
are summarized in Table 11.

Figure 16: Treatment Contract Role Hierarchy

Functions	Parameters		Notes
initialize	Input(s)		Initializes the contract after it is cloned from the implementation with the necessary information and set up the proper roles. This function would only be called once
	doctor_: address patient_: address medicine_: string form_: bytes32 dosage_: Dosage schedule_: Schedule		
	Return		
	none		
getPatient	Input(s)		Retrieves the address to the Patient contract, requires read access
	none		
	Modifier		
	canRead		
	Return		
	patient: address		

getMedicine	Input(s)	Retrieves the medicine in this Treatment contract
	none	
	Return	
	medicine_: string	
getTreatment	Input(s)	Retrieves the full information of this Treatment contract, requires read access
	none	
	Modifier	
	canRead	
	Return	
	medicine_: string form_: bytes32 dosage_: Dosage schedule_: Schedule isComplete_: bool	
edit	Input(s)	Makes update to this Treatment contract, require DOCTOR_ROLE
	medicine_: string (overload) form_: bytes32 (overload) dosage_: Dosage (overload) schedule_: Schedule (overload)	
	Modifier	
	onlyRole(DOCTOR_ROLE)	
	Return	
	none	
hasReadAccess	Input(s)	A private helper function for the canRead modifier
	none	
	Return	
	none	
isComplete	Input(s)	A private helper function for the getTreatment function to determine whether the treatment has been completed
	none	
	Return	
	bool	

Table 11: Treatment contract functions

4.2 Database Schema

Figure 17: Database Schema

The SQL database for this project includes three independent tables shown in Figure 16. The medicine table is for doctors to lookup when prescribing new treatments to patients. The log table is for patient reporting taking their medication for their treatments. To minimize the potential data exposure, only the report time and addresses of patient and treatment are stored in the SQL database. The subscription table holds the information necessary for the server to send push notification to users. Again, no details from the smart contracts are stored in the SQL database.

4.3 Server Side APIs

       This sections introduces the server side APIs in the Application server see in Table 12. We try to use Sign-in with Ethereum as much as possible to ensure user sign their request with their wallet. However, sometimes when the user isn't interacting with the application (e.g. the periodic check done by user's client), we rely on the session cookies.

Endpoints	Methods	Parameters	Notes
/signup	POST	address: address	Called after patient created their Patient contract, returns patient's contract address in json web token (JWT) format
/login	POST	message: object signature: string type: string	Login for patient and doctor, returns patient or doctor contract address in JWT format
/nonce	GET		Returns a nonce used for Sign-in with Ethereum message
/search_med	GET	search: string	Returns the results of the SQL query
/report	POST	message: object signature: string type: string	Patient (batch) reporting taking medication
/push	POST	address: address treatment: address access: address	Send push notification to doctors or patients for access requests or reminder for medication
/check	POST	address: address schedule: object	Patient's client periodically check whether patient is on track of the treatment
/subscribe	POST	subscription: object address: address	Save the browser endpoint of patient for push notification

Table 12: Server Side APIs

Chapter 5 Evaluation

This chapter will perform evaluation on this project. First we establish the two scenarios,

then we evaluate the robustness of the access control and the cost of the system proposed in this

report.

5.1 Workflow and Setup

This section will go through the workflow for patient and doctor and the setup for the

evaluation. We will simulate the authorization process to test the first scenario mentioned in

section 3.1, then prescribe 1000 treatments to evaluate the overall cost of the system.

5.1.1 Patient and Doctor Workflow

When a patient first make a doctor visit, they can fill out the new patient form illustrated

in Figure 18. The patient bring this smart contract to other healthcare providers or applications

with consistent information supported by the blockchain network. The patient can view the

profile and show the QR code of the contract address, illustrated in Figure 19, 20. For doctors,

their smart contracts are created by the administrator of the EHR system.

In the case of a doctor visit in which the doctor uses the medication reminder application,

the patient go to the application and sign in using SIWE and the application then request access

to the patient's contract, illustrated in Figure 21, 22, 23. Doctors can go through the same process

to sign in to the application. Patient and doctor then are greeted with their respective dashboard

shown in Figure 24, 25.

To prescribe a treatment to a patient, the doctor can enter patient's wallet address or scan

patient's QR code, illustrated in Figure 26. Doctor without access to patient's contract can

request access here, illustrated in Figure 27. The patient can update their profile and accept the

access request illustrated in Figure 28 Doctor with access can view patient's information and

proceed through the form to prescribe the treatment to the patient, illustrated in Figure 29, 30, 31,

32. Patient see the treatments in the dashboard and report after taking the medication illustrated

in Figure 33. Doctor can view the recent progress of a specific treatment and send reminder

accordingly, illustrated in Figure 34. Patient receives notification from the application or their

doctor if they fall behind in their treatment, illustrated in Figure 35.

New patient form

Please fill in this form to create your profile.

Name

Elliot Alderson

Gender

Male

Date of birth

09 / 17 / 1986

Height(cm)

175

Weight(kg)

70

Allergy(Select all that apply)

Food

☐ Balsam of Peru
☐ Buckwheat
☐ Celery
☐ Egg
☐ Fish
☐ Fruit
☐ Garlic
☐ Oats
☐ Maize
☐ Milk
☐ Mustard
☐ Peanut
☐ Poultry meat
☐ Rice
☐ Sesame
☐ Shellfish
☐ Tartrazine
☑ Tree nut
☐ Wheat

Medicine

☐ Tetracycline
☐ Dilantin
☐ Tegretol
☐ Penicillin
☐ Cephalosporins
☐ Sulfonamides
☐ Non-steroidal anti-inflammatories
☐ Intravenous contrast dye
☐ Local anesthetics

Other substance

☐ Do you drink alcohol?
☐ Do you smoke cigarette?
☑ Do you use cannabis?

By creating an account you agree to our Terms & Privacy.

Cancel   Complete

Figure 18: New patient form in Manager

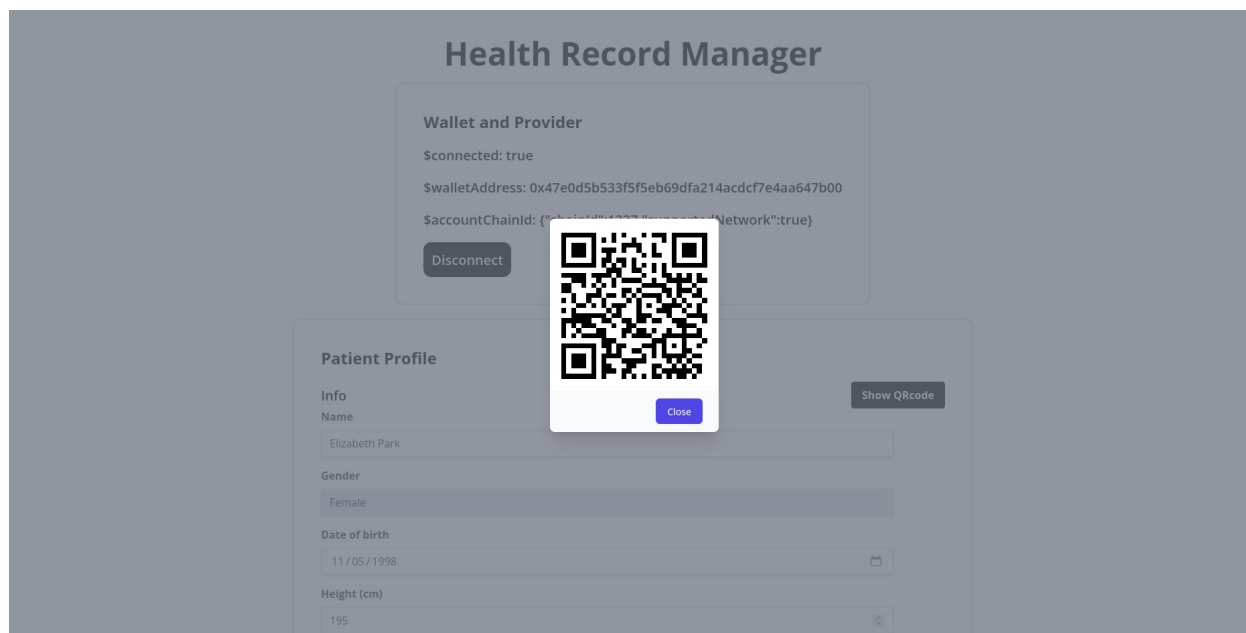**Figure 19: Patient profile page in Manager**

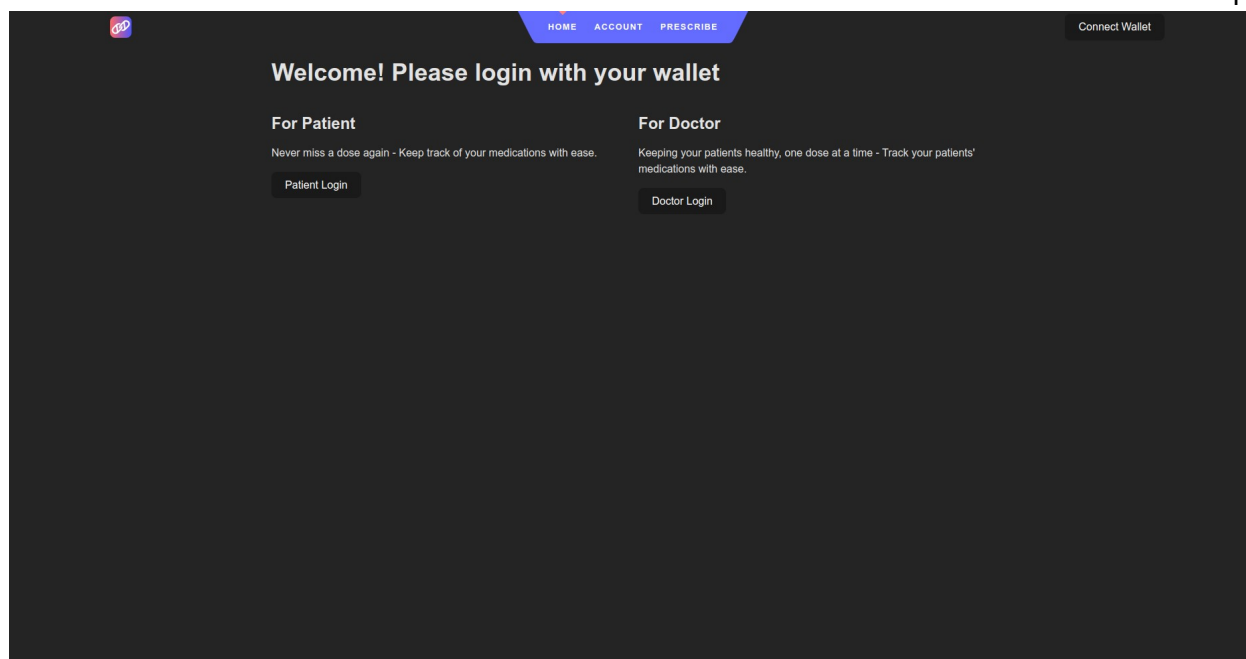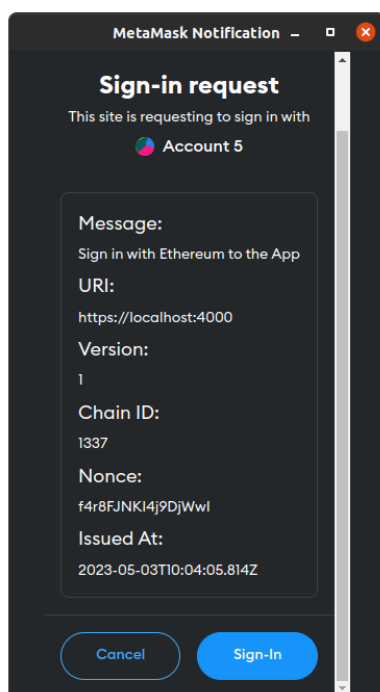**Figure 20: QR code for patient contract address in Manager**

Figure 21: Application landing page

Figure 22: Users sign in to

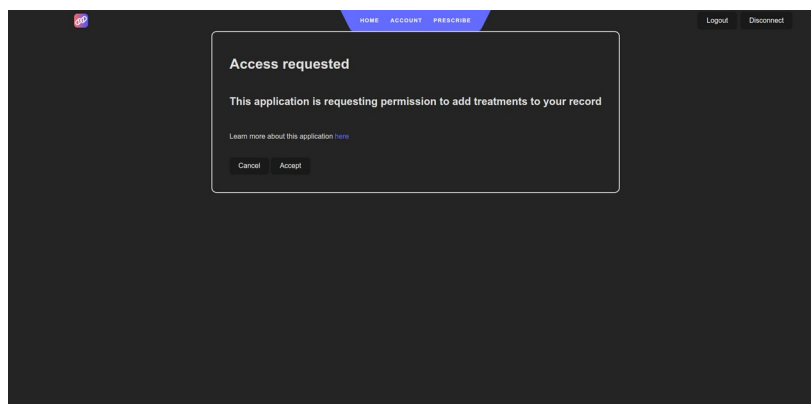**Application using SIWE**

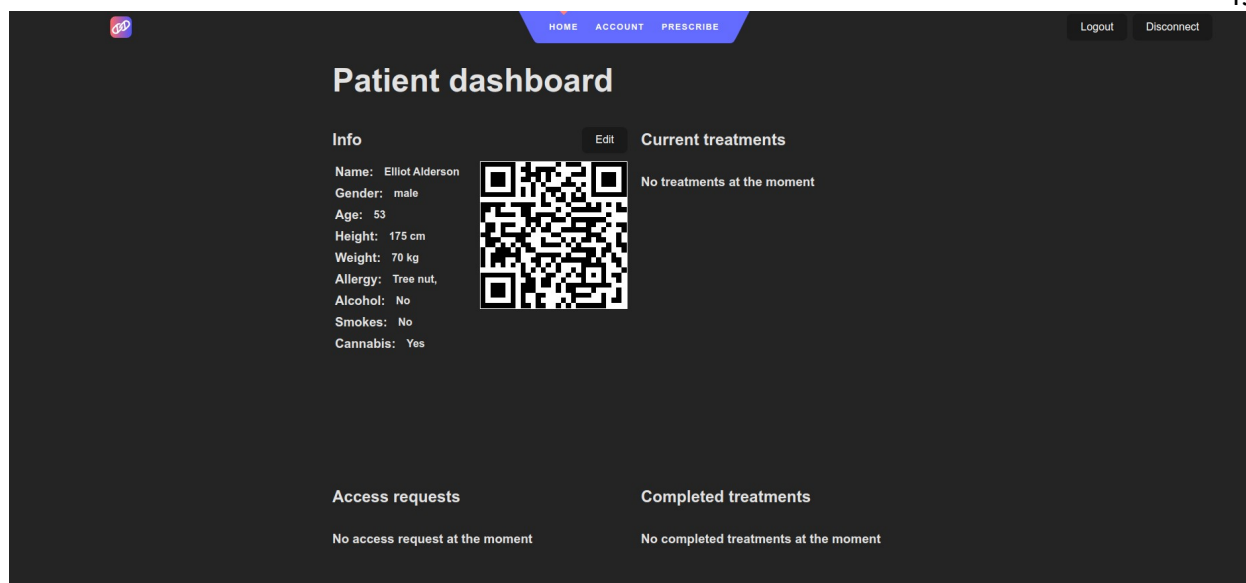**Figure 23: Application request access to Patient contract**

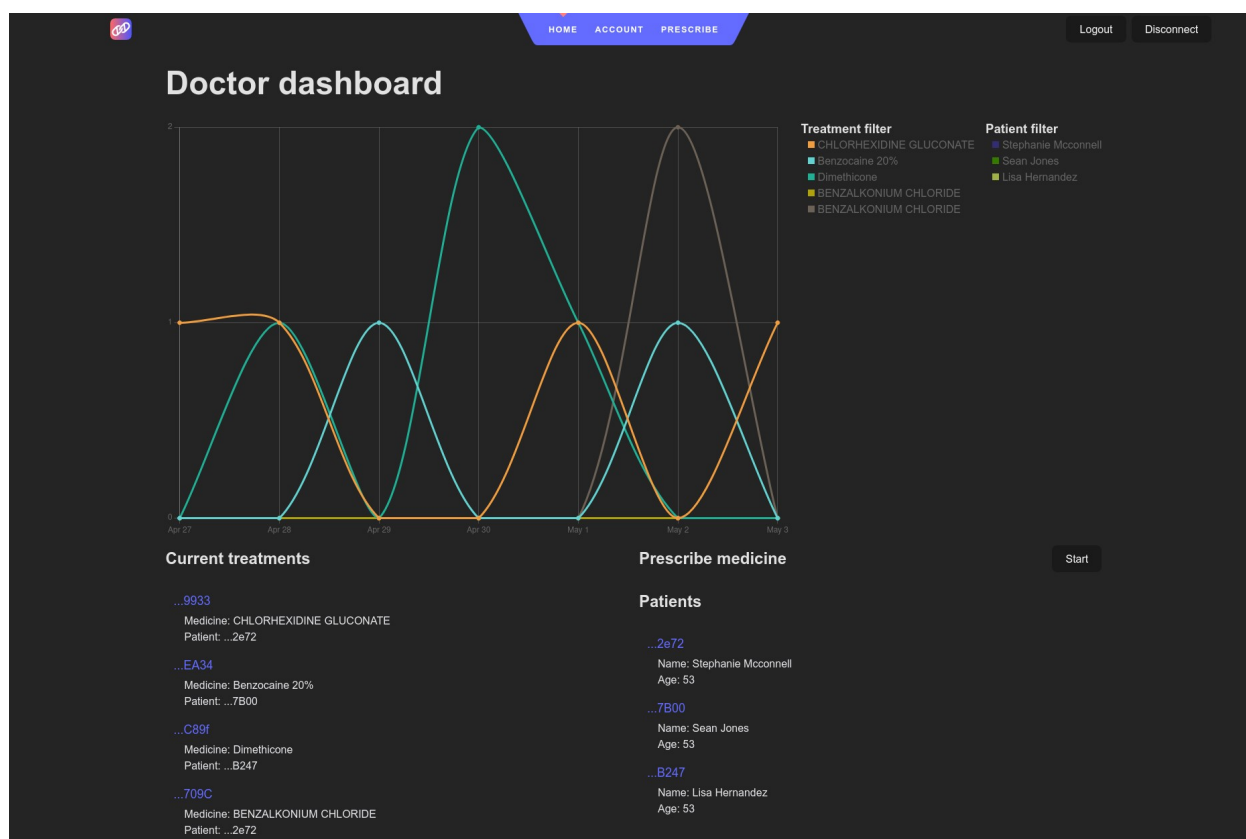**Figure 24: Patient dashboard in Application**

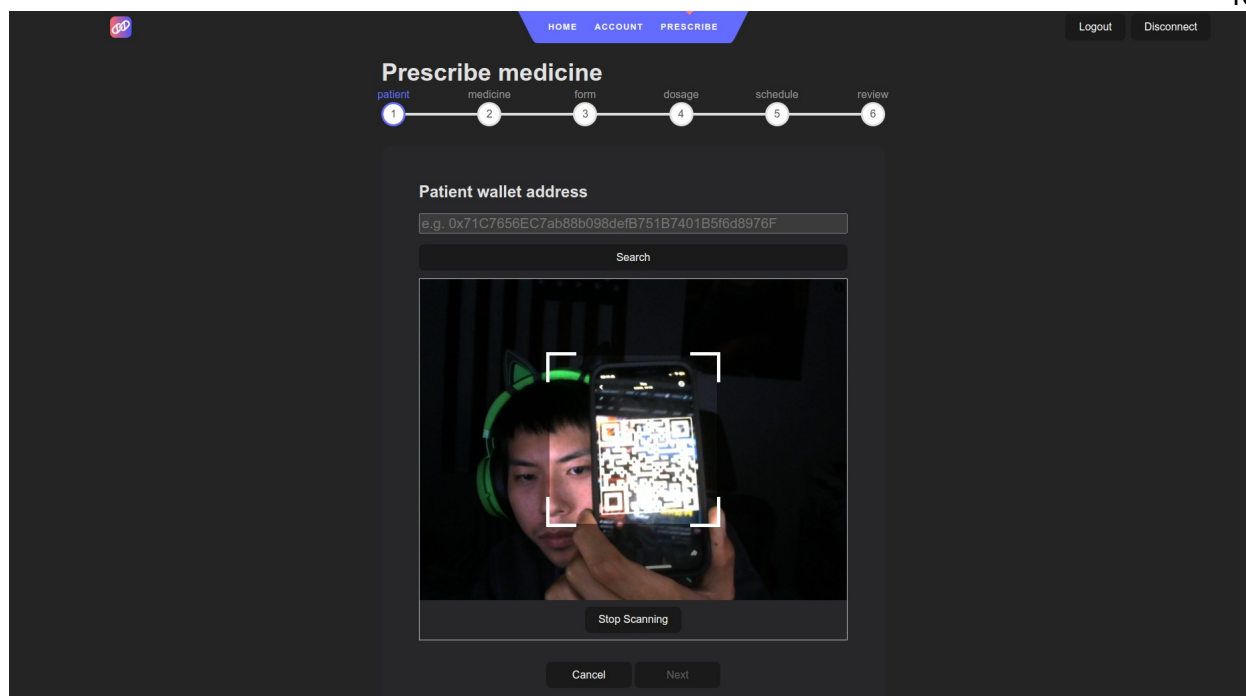**Figure 25: Doctor dashboard in Application**

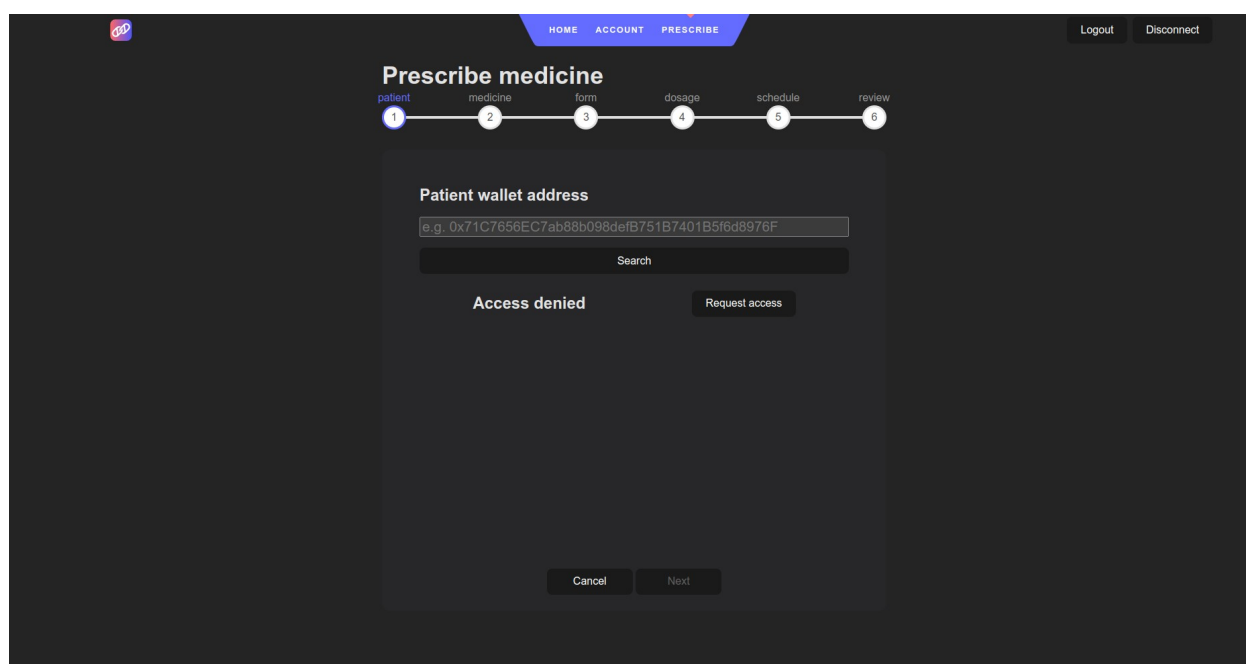**Figure 26: Doctor fetch a patient contract to prescribe a treatment**

Figure 27: Doctor without access request access from patient

Figure 28: Patient makes changes to profile and manages access request in Manager

Figure 29: Doctor with access receives patient info and proceed to prescribe treatment

Figure 30: Doctor searches for medication
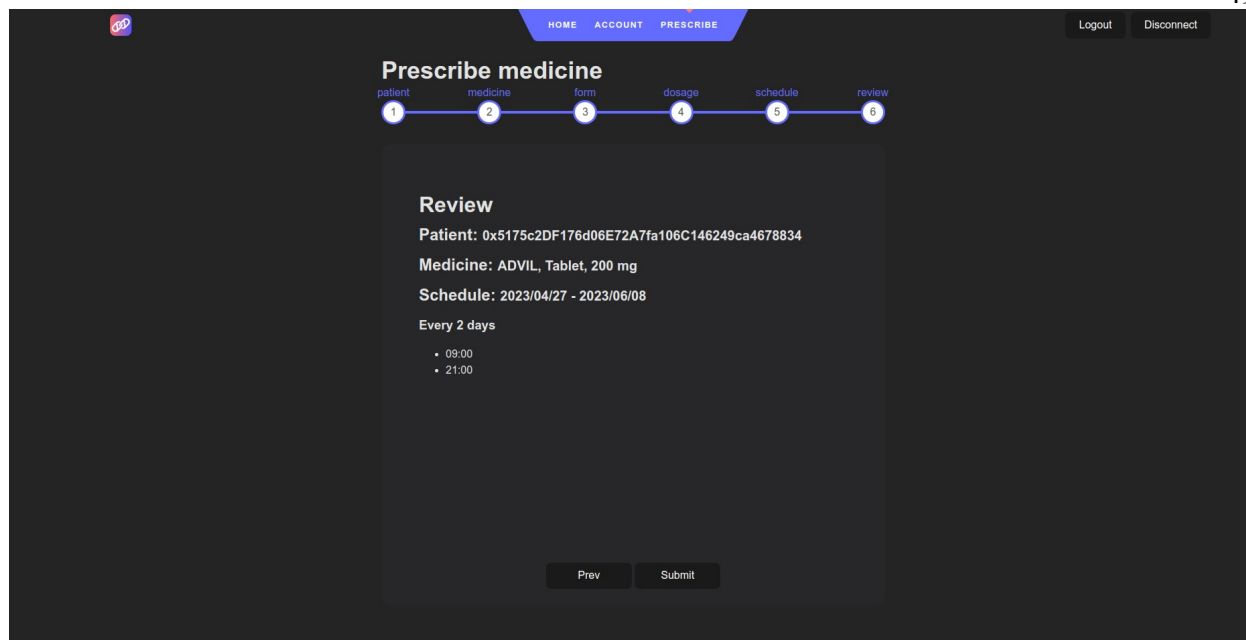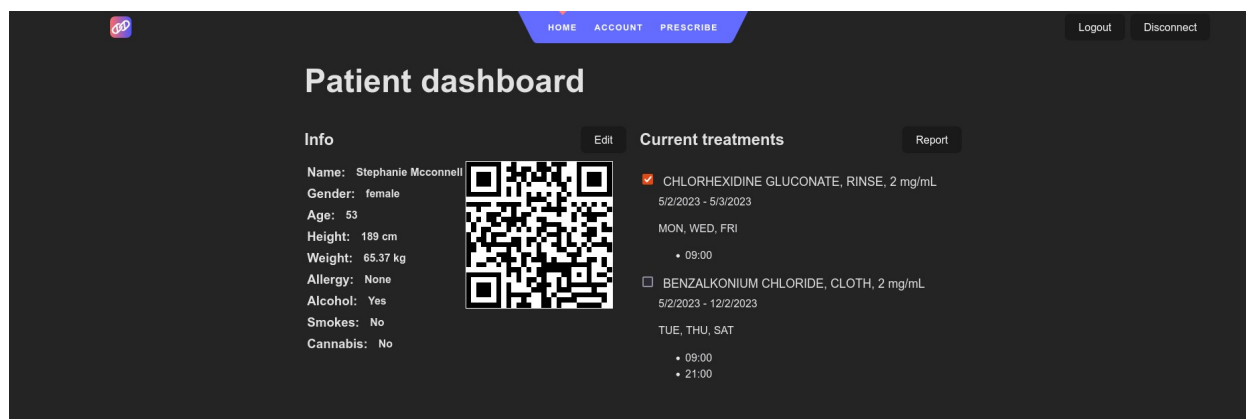
Figure 31: Doctor set schedule for treatment

Figure 32: Doctor review and submit treatment for patient

Figure 33: Patient select and report taking medication for treatment
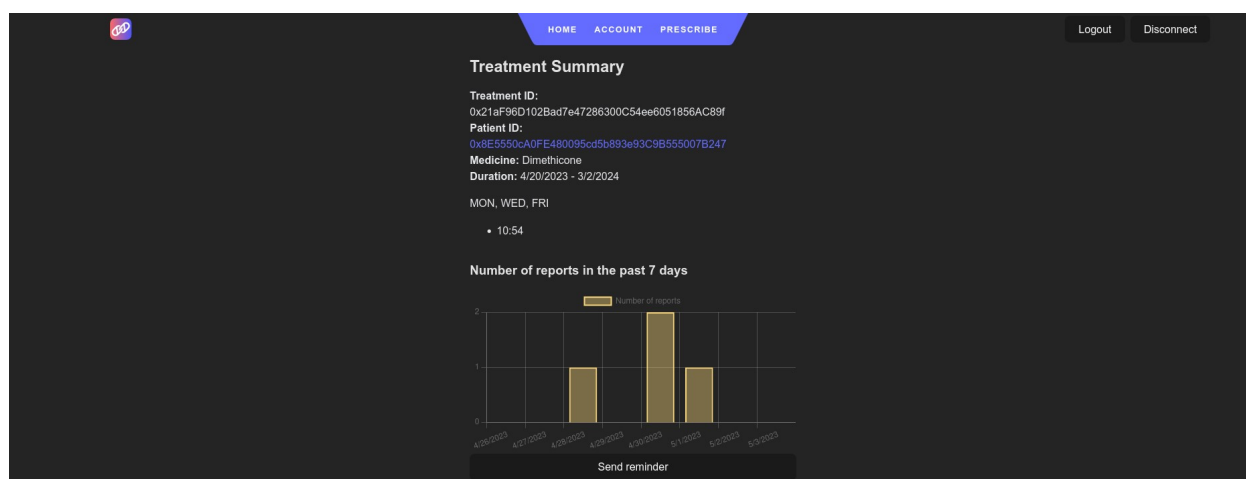
Figure 34: Doctor view treatment summary and recent progress
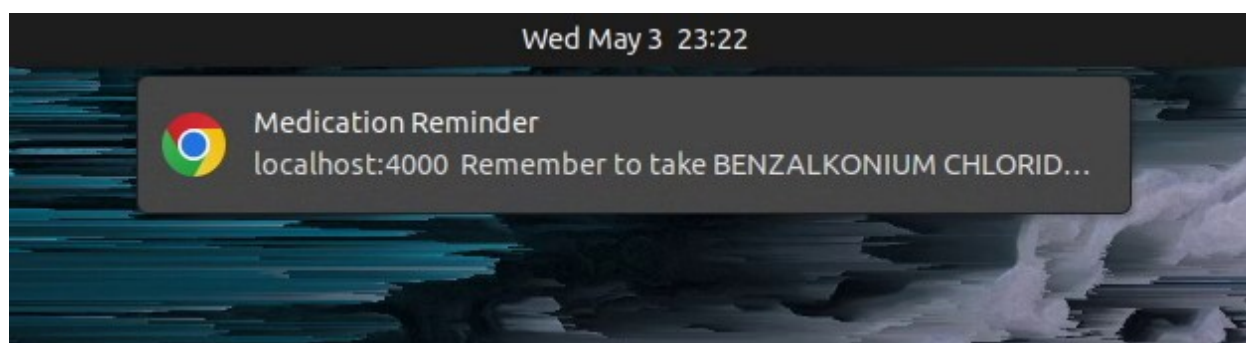
Figure 35: Medication reminder

5.1.2 Evaluation Setup

Since we have limited accounts to work with, we will setup all 10 accounts to be both doctor and patient. We have each account request the doctor role from each patient contract except for the pairs of accounts in Table 13.

Patient	Doctor
4	1
3	2
8	2
4	5
3	8
4	9

Table 13: Access Request Exception

Next we have each accounts accept all the access requests except for the accounts in Table 14 ignoring the requests, which results in only doctor 0 receiving the doctor role for all patients. Lastly, we have doctor 0 and doctor 1 attempt to accept the access requests ignored in Table 14 on the patients' behalf. After the authorization process, we will use each doctor accounts to check their access to each patient contract in this RBAC system. The doctor role requests and patients accepting requests are described in Table 15.

Patient	Doctor
2	3
2	4
2	5
2	6
2	7
2	8
2	9
5	3
5	8
6	4
7	5
8	3
8	6
9	7

Table 14: Ignore Access Request

Patient	Doctor Role Requests from	Accepted
0	1,2,3,4,5,6,7,8,9	1,2,3,4,5,6,7,8,9
1	0,2,3,4,5,6,7,8,9	0,2,3,4,5,6,7,8,9
2	0,1,3,4,5,6,7,8,9	0,1
3	0,1,4,5,6,7,9	0,1,4,5,6,7,8,9
4	0,2,3,6,7,8	0,2,3,6,7,8
5	0,1,2,3,4,6,7,8,9	0,1,2,4,6,7,9
6	0,1,2,3,4,5,7,8,9	0,1,2,3,5,7,8,9
7	0,1,2,3,4,5,6,8,9	0,1,2,3,4,6,8,9
8	0,1,3,4,5,6,7,9	0,1,4,5,7,9
9	0,1,2,3,4,5,6,7,8	0,1,2,3,4,5,6,8

Table 15: Doctor Role Requests and Acceptances

After testing the access control, we proceed to run a stress test of prescribing 1,000 treatments to across all 10 accounts. Then we will look at the overall cost of the system performing major functionalities.

5.2 Results

5.2.1 Access Control

5.2.1.1 Patient Contract Read and Write Access

Table 16 shows the resulting read and write access on each patient contract. In the table, each row represent patient contract; and each column represent doctor account. In each column, first cell represents read access, second cell represents write access. Green means the user has access, and red means user does not have access. We can first see all the diagonal cells (P0, D0), (P1, D1), etc. They all have read and write access, since they all have the patient role to their own Patient contract. Patient 0 and patient 1 both accepted doctor role request from all users, so D0 – D9 all have read and write access to P0 and P1. Patient 2 only accepted doctor 0 and doctor 1 requests, so D3 – D9 do not have read and write access to P2. Then doctor 2 and doctor 8 did not request doctor role from P3, so those two doctors do not have read and write access to P3. Doctor 1, doctor 5 and doctor 9 did not request doctor role from patient 4, so D1, D5 and D9 do not have read and write access to P4. Doctor 3 and doctor 8 both requested doctor role from patient 5, but patient 5 did not accept the request, so D3 and D8 do not have access to P5, and so on. Despite having doctor 0 and doctor 1 attempt to accept the doctor requests for the other patients, it was unsuccessful, because only account with patient role can grant doctor role to other accounts. The results in Table 16 match the expected results in Table 15. Doctors that did

not request access to patients' contract do not have read or write access to those contracts.

Doctors that were ignored by patients also do not have read or write access to those contracts.

	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
P0	green	green	green	green	green	green	green	green	green	green
P1	green	green	green	green	green	green	green	green	green	green
P2	green	green	green	red	red	red	red	red	red	red
P3	green	green	red	green	green	green	green	green	red	green
P4	green	red	green	green	green	red	green	green	green	red
P5	green	green	green	red	green	green	green	green	red	green
P6	green	green	green	green	red	green	green	green	green	green
P7	green	green	green	green	green	red	green	green	green	green
P8	green	green	red	red	green	green	red	green	green	green
P9	green	green	green	green	green	green	green	red	green	green

Table 16: Patient Contract Read and Write Access Table, green means has access, red

means no access

5.2.1.2 Application-level Access to Prescribe

Table 17 shows the results of access control test in the Application. The results are mostly the same as Table 16 except for the diagonal cells. Since each patient do not have doctor role to their own contract, they can read and write but not prescribe treatment to themselves. The results shows that the role-based-access-control is viable in this use case. Ideally, before granting an application the application role, the user should examine the source code of the application, which would be publicly available on Etherscan. Since not every person has the knowledge to audit smart contract source code, organizations could be created to audit smart contracts and publish the results.

	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
P0	red	green	green	green	green	green	green	green	green	green
P1	green	red	green	green	green	green	green	green	green	green
P2	green	green	red	red	red	red	red	red	red	red
P3	green	green	red	red	green	green	green	green	red	green
P4	green	red	green	green	red	red	green	green	green	red
P5	green	green	green	red	green	red	green	green	red	green
P6	green	green	green	red	red	green	red	green	green	green
P7	green	green	green	green	red	green	red	red	green	green
P8	green	green	red	red	green	green	red	green	red	green
P9	green	green	green	green	green	green	green	red	green	red

Table 17: Application Prescribe Access Table, green means has access, red means no access

5.2.2 Cost

Because reading from the blockchain does not consume any gas, reading patients' profile, sending notification, and other read functions would not cost any in Ether. This section will evaluate the cost of the following functions:

- Add a patient

- Add a doctor

- Request access to a patient

- Accept an access request

- Prescribe a treatment

To run the cost tests, we opt to deploy our project to the Sepolia testnet [30]. Sepolia is one the first public testnets that switched to a PoS consensus matching the Ethereum main net. Sepolia also has similar block time to Ethereum main net, approximately 12 seconds. However, the gas fee on Sepolia is much lower than on Ethereum, since it is a much younger chain. For

calculation purposes, we will adopt the average gas price of 20 Gwei and average Ether price of

$1300 observed in January 2023 [27][29]. The formulas for calculation are as followed:

$$Cost(Ether) = Gas\,used \times Gas\,price \times 10^{-9}$$

$$Cost(USD) = Cost(Ether) \times Ether\,price$$

Cost (USD) and Gas Used for each Function. Color shows details about Cost (USD) and Gas Used.
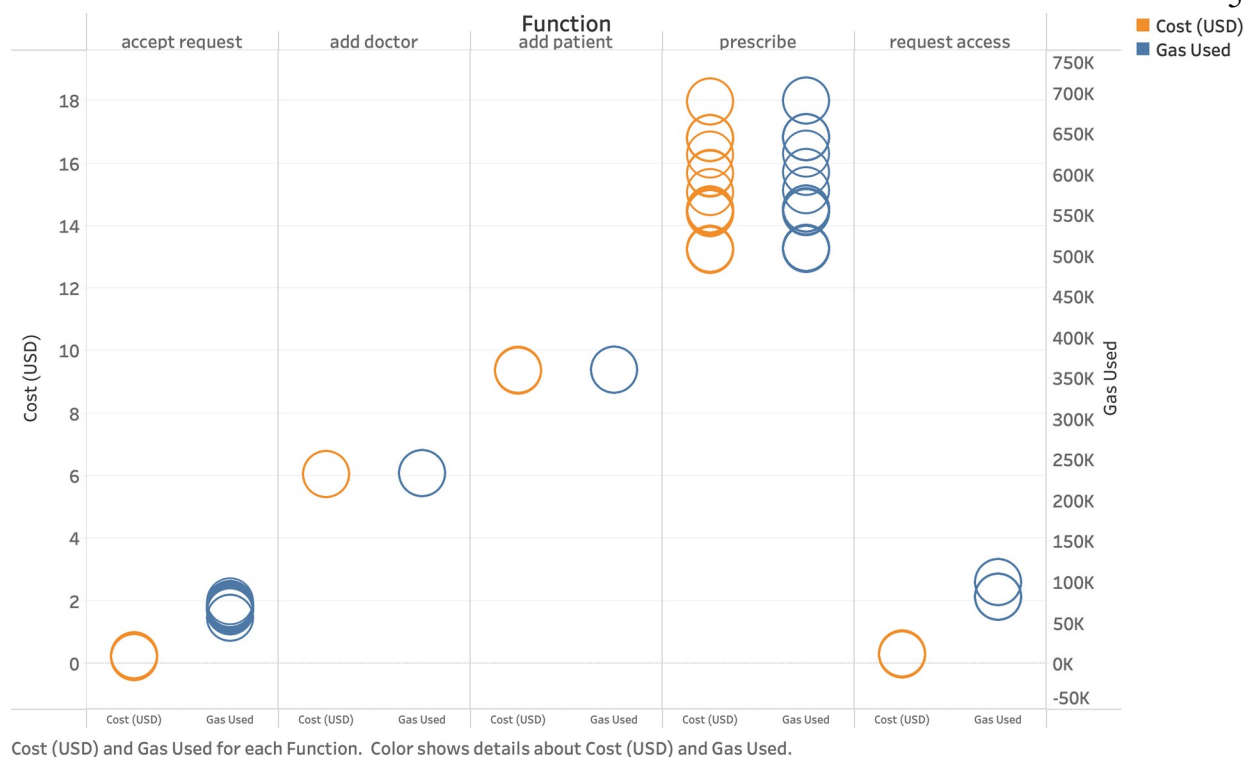
Figure 36: Cost (USD) and Gas Consumption

Figure 36 shows the gas used for each function and how it translate to USD. We can see that deploying new patient and doctor contracts consumed relatively consistent amount of gas, since they take in about the same amount of data for each deployment. Requesting access and accepting requests fluctuate a little bit in gas consumption, since both functions traverse the request queue in the patient contract to a certain degree. Both function could consume more or less gas depending on how long the request queue is. Lastly, prescribing new treatments has the biggest range in gas consumption, mostly due to the variations of treatment schedules and medicine string length.

Function	Average Gas Used	Average Cost (Ether)	Average Cost (USD)
Add patient	360,328	0.00720656	9.37
Add doctor	232,777	0.00465554	6.05
Request access	83,769	0.00020942	0.27
Accept request	67,652	0.00016913	0.22
Prescribe	569,617	0.01139233	14.81

Table 18: Average Cost of Five Tested Functions

Table 18 shows the average cost of the five tested functions. Based on the result, we can see that majority of the cost for patients originates from deploying the patient contract, which is fairly affordable at just under $10. With most people having a primary provider, the number of times a patient need to accept access request are kept to minimum. Doctor, on the other hand, has the heavier cost of $14.81 on average to prescribe treatments to patients. Depending on how frequent the doctor prescribe new treatments, it can be costly in the long run.

Despite the cost, a well funded hospital should be able to afford the $14.81 per prescription, given various revenues source and the money saved on server cost migrating to a blockchain-based EHR system. To further lower the cost to operate, the healthcare industry can deploy the EHR system on the younger and cheaper Polygon blockchain or even create its own blockchain.

Chapter 6 Conclusion and Future Work

6.1 Conclusion

In this project, we demonstrated that an interoperable and secure EHR system can be implemented with the help of blockchain technology. This system allows patients to take control of their consistent health data with tamper-proof access control. Healthcare providers and other applications can simply request access from patients when they need the data. This system is a solution to scenarios where sharing healthcare data while preserving privacy is the main concern, such as the two scenarios we mentioned in Chapter 3. With blockchain's transparency characteristic, patients can audit the smart contract of the application before granting access.

6.1.1 Limitation

Chapter 2 established that ever since Ethereum switched to PoS consensus, the block time has been consistently at 12 seconds, which means every test, prescription, diagnosis, etc. takes 12 seconds to be added to the health records. While an open sourced EHR system improves the overall interoperability, propagating information throughout the system takes longer than a off-chain solution.

Additionally, Blockchain's transparency brings the issue of information transmitted during contract calls are public. Although access control can block out unauthorized requests programmatically, the data in transactions can easily be viewed on Etherscan even when TLS is in placed, illustrated in Figure 37. Although encryption is computational intensive tasks that are not suitable for on-chain operation, there are companies such as Zama working on homomorphic encryption solutions that allows computation to be performed directly on encrypted data [43].
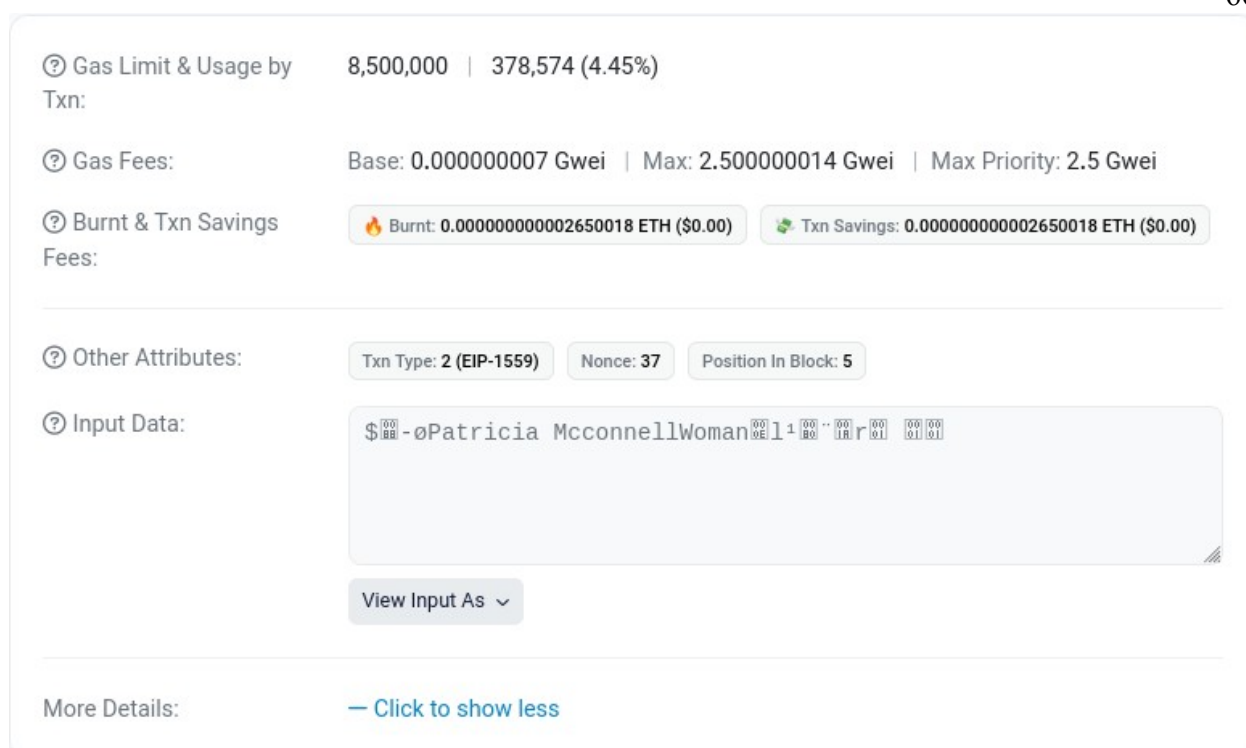
Figure 37: Transaction Details on Etherscan

6.1 Future Work

This project can be further improved in terms of the features of the EHR and the robustness of access control, for example, allowing emergency contacts to grant access request when the patient is impaired to do so, or implement relationship-based access control (ReBAC) to define more fine-grained permissions on parts of the health records. This project currently focuses on allowing patients to own and share consistent health records, which means the patients will have to make the judgment of who they want to share the health records with. This system assumes that patients are not prone to social engineering, but that is not the case in the real world. Approximately 20% of data breach cases involved social engineering and a record 1.2

millions phishing attacks observed in 2022 [37][40]. Without proper awareness training, patients can be tricked into granting cybercriminals access to their health records.

Storing EMRs on distributed system also faces challenges from legislation. Aside from the technical compliance on handling healthcare data, certain practices of storing data on blockchain may fall under anti-trust behaviors [38]. As mentioned in chapter 5, developers can build a new alternative chain from Ethereum to lower the cost of gas, but doing so could pose a membership for healthcare providers. Such membership can deny healthcare providers from requesting access all together. Therefore, in the future we want to continue to improve features and security of EHR on blockchain, as well as the legislation on this practice. That way a blockchain-based EHR system can become a viable alternative to the current implementation.

References

[1] "Electronic Health Record Systems," *HSS.gov*, Feb. 13, 2020.

https://www.hhs.gov/sites/default/files/electronic-health-record-systems.pdf (accessed Jan.

06, 2023).

[2] N. P. Terry, "'Will the Internet of Things Transform Healthcare?' by Nicolas P. Terry,"

Scholarship@Vanderbilt Law. https://scholarship.law.vanderbilt.edu/jetlaw/vol19/iss2/5

(accessed Jan. 19, 2023).

[3] N. Terry, "Existential challenges for healthcare data protection in the United States," *Ethics,*

Medicine and Public Health, no. 1, pp. 19–27, Jan. 2017, doi: 10.1016/j.jemep.2017.02.007.

[4] E. Li, J. Clarke, H. Ashrafian, A. Darzi, and A. L. Neves, "The Impact of Electronic Health

Record Interoperability on Safety and Quality of Care in High-Income Countries: Systematic

Review," *Journal of Medical Internet Research*, no. 9, p. e38144, Sep. 2022, doi:

10.2196/38144.

[5] Angst C, Agarwal R, Downing J. An empirical examination of the importance of defining

PHR for research and for practice. Robert H. Smith School Research Paper No. RHS-06-

011; 2006.

[6] Ermakova T, Fabian B, Zarnekow R. Security and Privacy System Requirements for

Adopting Cloud Computing in Healthcare Data Sharing Scenarios. Proceedings of the 19th

Americas Conference on Information Systems, 2013.

[7] Kuo K-M, Ma C-C, Alexander J. How do patients respond to violation of their information

privacy. Health Information Manag J 2013;43(2):23–33.

[8] E. V. Bernstam *et al.*, "Quantitating and assessing interoperability between electronic health records," *Journal of the American Medical Informatics Association*, no. 5, pp. 753–760, Jan. 2022, doi: 10.1093/jamia/ocab289.

[9] "More than 20 Serious Vulnerabilities in OpenEMR Platform Patched," HIPAA Journal. [Online]. Available: https://www.hipaajournal.com/more-than-20-serious-vulnerabilities-in-openemr-platform-patched/. [Accessed: May 17, 2023].

[10] "Summary of the HIPAA Security Rule | HHS.gov," *HHS.gov*, Nov. 20, 2009. http://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html (accessed Jan. 19, 2023).

[11] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). [Available at: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L .2016.119.01.0001.01.ENG&toc=OJ:L:2016:119:TOC].

[12] Medisafe. [Online]. Available: https://medisafeapp.com/. [Accessed: May 17, 2023].

[13] "Track your medications on iPhone," Apple Support. [Online]. Available: https://support.apple.com/guide/iphone/track-your-medications-iph811670c81/ios. [Accessed: May 17, 2023].

[14] "Access Control - OpenZeppelin Docs," *Documentation - OpenZeppelin Docs*. https://docs.openzeppelin.com/contracts/4.x/access-control (accessed Jan. 13, 2023).

[15] "authentication - Glossary | CSRC," *NIST Computer Security Resource Center | CSRC*. https://csrc.nist.gov/glossary/term/authentication (accessed Jan. 19, 2023).

[16] "authorization - Glossary | CSRC," *NIST Computer Security Resource Center | CSRC*. https://csrc.nist.gov/glossary/term/authorization (accessed Jan. 17, 2023).

[17] US Department of Health and Human Services. Enforcement results by year. [Available at: http://www.hhs.gov/hipaa/for-professionals/compliance-enforcement/data/enforcement-results-by-year/index.html].

[18] M. J. Mabee, "Healthcare Data Breaches in South Dakota: Post-Breach Legislation Is Not Enough," *South Dakota Law Review, vol. 65, no. 3*, 2020, Accessed: Jan. 13, 2023. [Online].

[19] D. B. Lafky and T. A. Horan, "Personal health records," *Health Informatics Journal*, no. 1, pp. 63–71, Mar. 2011, doi: 10.1177/1460458211399403.

[20] T. Li and T. Slee, "The effects of information privacy concerns on digitizing personal health records," *Journal of the Association for Information Science and Technology*, no. 8, pp. 1541–1554, Apr. 2014, doi: 10.1002/asi.23068.

[21] M. A. de Carvalho Junior and P. Bandiera-Paiva, "Strengthen Electronic Health Records System (EHR-S) Access-Control to Cope with GDPR Explicit Consent," *Journal of Medical Systems*, no. 10, Aug. 2020, doi: 10.1007/s10916-020-01631-5.

[22] "Transport Layer Security (TLS) - Glossary | CSRC," *NIST Computer Security Resource Center | CSRC*. https://csrc.nist.gov/glossary/term/transport_layer_security (accessed Jan. 19, 2023).

[23] V. Buterin, "A next-generation smart contract and decentralized application platform," *white paper*, Jan. 2014, Accessed: Jan. 20, 2023. [Online].

[24] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," Journal of Cryptology, no. 2, pp. 99–111, Jan. 1991, doi: 10.1007/bf00196791.

[25] P. Murray, N. Welch, and J. Messerman, "EIP-1167: Minimal Proxy Contract," *Ethereum Improvement Proposals*, Jun. 22, 2018. https://eips.ethereum.org/EIPS/eip-1167 (accessed Jan. 13, 2023).

[26] W. Chang, G. R. Rocco, B. M. Millegan, and N. Johnson, "EIP-4361: Sign-In with Ethereum," *Ethereum Improvement Proposals*, Oct. 11, 2021. https://eips.ethereum.org/EIPS/eip-4361 (accessed Jan. 13, 2023).

[27] "Ethereum Gas Tracker | Etherscan," *Etherscan*. https://etherscan.io/gastracker (accessed Jan. 13, 2023).

[28] "Pool Stats - BTC.com," *BTC.com First website for blockchainers*. https://btc.com/stats/pool (accessed Jan. 13, 2023).

[29] "Ethereum Price | ETH Price Index and Live Chart- CoinDesk," *CoinDesk: Bitcoin, Ethereum, Crypto News and Price Data*. https://www.coindesk.com/price/ethereum/ (accessed Jan. 13, 2023).

[30] "TESTNET Sepolia (ETH) Blockchain Explorer," *Ethereum (ETH) Blockchain Explorer*. https://sepolia.etherscan.io/ (accessed Jan. 13, 2023).

[31] "Proof-of-stake (PoS) | ethereum.org," *ethereum.org*. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (accessed Jan. 13, 2023).

[32] "Proof-of-work (PoW) | ethereum.org," *ethereum.org*. https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/ (accessed Jan. 13, 2023).

[33] "Introduction to smart contracts | ethereum.org," *ethereum.org*.

https://ethereum.org/en/developers/docs/smart-contracts/ (accessed Jan. 20, 2023).

[34] "Proxies - OpenZeppelin Docs," *Documentation - OpenZeppelin Docs*.

https://docs.openzeppelin.com/contracts/4.x/api/proxy#minimal_clones (accessed Jan. 13,

2023).

[35] OpenZeppelin, "Cheap Contract Deployment Through Clones," Mar. 02, 2021.

https://www.youtube.com/watch?v=3Mw-pMmJ7TA (accessed Jan. 13, 2023).

[36] "OWASP Top Ten | OWASP Foundation," *OWASP Foundation, the Open Source*

Foundation for Application Security | OWASP Foundation. https://owasp.org/www-project-

top-ten/ (accessed Jan. 13, 2023).

[37] "2022 Data Breach Investigations Report | Verizon," Verizon Business.

https://www.verizon.com/business/en-gb/resources/reports/dbir/ (accessed Jan. 16, 2023).

[38] C. S. Hutchinson and M. A. Egorova, "Potential Legal Challenges for Blockchain

Technology in Competition Law," *Baltic Journal of Law & Politics*, no. 1, pp. 81–107,

Jun. 2020, doi: 10.2478/bjlp-2020-0004.

[39] "Solidity 0.8.6 documentation," *Solidity — Solidity 0.8.17 documentation*.

https://docs.soliditylang.org/en/v0.8.6/introduction-to-smart-contracts.html?

highlight=delegatecall#delegatecall-callcode-and-libraries (accessed Jan. 16, 2023).

[40] "Phishing Activity Trends Report Q3 2022," APWG | Unifying The Global Response To

Cybercrime, Dec. 12, 2022. PHISHING ACTIVITY TRENDS REPORT (accessed Jan. 17,

2023).

[41] R. S. Sandhu, E. J. Coyne, H. L. Feinstein and C. E. Youman, "Role-based access control models," in Computer, vol. 29, no. 2, pp. 38-47, Feb. 1996, doi: 10.1109/2.485845.

[42] "Role-Based Access Controls | CSRC," *NIST Computer Security Resource Center | CSRC*. https://csrc.nist.gov/publications/detail/conference-paper/1992/10/13/role-based-access-controls (accessed Jan. 17, 2023).

[43] "Zama – Fully homomorphic encryption," Zama AI. https://www.zama.ai/. Accessed: May 16, 2023.