19N720 - PROJECT WORK 1

Privacy-Preserving Encrypted Medical Record Management Using Blockchain

Guide: Ms. S.K.Abirami

Harini M	**- 22N219**
Pavithra E	**- 22N236**
Priyadharshini M	**- 22N240**
Selvaraju Nikethna Sri	**- 22N249**
Subiksha S	**- 22N259**

BACHELOR OF ENGINEERING

Branch: COMPUTER SCIENCE AND ENGINEERING

(Artificial Intelligence and Machine Learning)

Of Anna University

NOVEMBER 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641004

TABLE OF CONTENTS PAGES

ACKNOWLEDGEMENT

We would like to extend our sincere thanks to our Principal In-charge, Dr. K. Prakasan, for providing us with this opportunity to develop and complete our project in our field of study.

We express our sincere thanks to our Head of the Department, Dr. G.Sudha Sadasivam, for encouraging and allowing us to present the project at our department premises for the partial fulfillment of the requirements leading to the award of BE degree.

We take immense pleasure in conveying our thanks and a deep sense of gratitude to Ms.S.K.Abirami, the Internal Project Mentor and Dr.G.R.Karpagam, the Programme Coordinator, Department of Computer Science and Engineering, for her valuable inputs, guidance, encouragement, wholehearted cooperation, and constructive criticism throughout the duration of our project. Also we would like to take this opportunity to place my sincere thanks to our Tutor Dr.D.Indumathi for providing motivation and guidance.

ABSTRACT

As hospitals increasingly adopt the use of digital healthcare systems, there poses a privacy and security risk for patient data. Patients have little control over how their data is used or accessed within a hospital. This project aims to address these concerns utilizing blockchain, making patients the ultimate owner of their data.

In this system, medical records are encrypted and stored securely while consent for data sharing is granted by smart contracts located on a blockchain. Patients have the ability to determine access to certain aspects of their information and achieve privacy and trust with each transaction. The system also supports safe redaction and updates while still maintaining the integrity of the records, in order to meet data protection requirements such as the GDPR.

In summary, this project articulates how a strong encryption model, built on the principles of blockchain technology and patient consent to access, can provide a secure, transparent, patient-centric healthcare framework.

CHAPTER-1
INTRODUCTION

Chapter 1 deals with the introduction of the project. It describes the problem statement, project objectives and the overall outline of the project work.

1.1 PROBLEM STATEMENT

Patients often visit different hospitals and clinics for treatment, with each organization having its record of the patient's medical information. Because the systems are uncoordinated, the patient must disclose their personal and medical information to each hospital, making it more likely for the patient's information to be exposed and exploited. Patients cannot achieve unified control over their information, making it impossible to manage the approvals to access their information or update or remove deprecated information. This lack of control leads to privacy and data protection compliance concerns.

1.2 OBJECTIVES

The primary objectives of the project are to,

- Create a blockchain-enabled system that encrypts and stores medical records securely off-chain.
- Provide redaction functionality through chameleon hash functions without compromising the immutability principle of blockchain.
- Support selective claim verification using zero-knowledge proofs (e.g., proving age is 18 without revealing any other information).
- Develop a user-friendly interface, managing role-based access control for patients, doctors and administrators.

1.3 STAKEHOLDERS OF THE PROJECT

The primary users of the proposed system are:

1. **Patient:** The main user of the system who owns and controls their medical data. Patients can upload, manage, and selectively allow access to their medical records with other authorized users of the system.
2. **Doctor / Healthcare Professional:** A medical person who is granted access to a patient's medical records for diagnosis or treatment purposes, after consent from the patient.

3. **Hospital / Healthcare Institution:** A service provider that stores and retrieves medical records when authorized. The hospital may also verify records to ensure compliance with healthcare regulations.

1.4 SCOPE OF THE PROJECT

The current phase of the project focuses on handling the patient data securely within a single hospital's healthcare system. The system primarily leveraged the immutability in blockchain and specific security algorithms like chameleon hashes and zero knowledge proofs. The integration between multiple hospitals using federated or swarm learning is the future extension that is planned for Project Work - II.

Tasks related to hospital administration or medical diagnosis or appointment scheduling is not within the defined scope of the project.

CHAPTER-2
LITERATURE SURVEY

Chapter 2 deals with the survey of the existing works in the scope of the project work. It presents the key takeaways from different research papers and articles.

2.1 INTRODUCTION

Since the introduction of Blockchain technology in 2008, blockchain applications have mostly been researched in the realm of cryptocurrencies and, more recently, in terms of healthcare data. Traditional systems encounter barriers such as fragmented records, limited patient control and access to their records, and significant data breaches. The decentralization and tamper proof nature of Blockchain (and health records in particular) offers a way for patients to have ownership of their medical data.

This survey reviews these works to understand current approaches and support the design of the proposed Decentralized EHR Management System.

2.2 REVIEW OF EXISTING WORK

2.2.1 Blockchain for medical record management

The "Blockchain Technology for Electronic Health Records" by Yujin Han and two others discusses the advantages and disadvantages of applying blockchain technology in the management of electronic health records. It discusses the benefits of immutability, decentralization, and privacy principles present in blockchain, while also referencing the drawbacks of inefficiency and high energy requirements for the consensus in blockchain.

The "Blockchain-Based Healthcare Records Management Framework: Enhancing Security, Privacy, and Interoperability" paper by Noor Ul Ain Tahir and 5 others proposes a EHR framework with a unique hyper ledger-enabled management system that outperforms other similar works (MediChain, CharmHealth) with its minimal response time. This work utilizes IPFS, smart contracts and Hardhat platform for its implementation.

2.2.2 Chameleon Hashes

"Chameleon Hashing and Signatures" paper by Hugo Krawczyk and Tal Rabin introduced the concept of chameleon hash function as a trapdoor collision-resistant hash function. Without the trapdoor, it behaves like a standard collision-resistant hash. With the trapdoor, collisions can be efficiently found for any given hash output.

“Redactable Blockchain From Decentralized Chameleon Hash Functions” by Meng Jia and 5 others mentions how redaction in blockchain is possible using chameleon hash functions. This work also proves that redaction is efficient this way compared to the traditional approaches.

2.2.3 Zero Knowledge Proofs

“The Knowledge Complexity Of Interactive Proof Systems” by Shafi Goldwasser, Silvio Micali, And Charles Rackoff in February 1989 introduces zero knowledge proofs as those proofs that convey no additional knowledge other than the correctness of the proposition in question.

The “Succinct Non-Interactive Arguments via Knowledge of Exponent Assumptions” by Jens Groth presents Groth16, one of the most efficient constructions of zk-SNARKs. The scheme allows proving that a computation is correct without revealing inputs, using pairing-based cryptography on elliptic curves. Groth16 dramatically reduces proof size and verification time, making it suitable for real-world systems like Zcash.

The “PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge” by Ariel Gabizon and 3 others introduces a universal and updatable zk-SNARK that improves flexibility and setup reusability compared to Groth16. It uses polynomial commitment schemes (based on Kate commitments) and a permutation argument to verify complex arithmetic circuits efficiently.

2.3 CONTEXT OF OUR WORK

The above section provides a complete overview of different technologies chosen for the proposed system. While there exists solutions utilizing blockchain technology in EHR management, most of them highlight secure data storage or immutability. None of them essentially focuses on a complete patient-centric approach, where individuals control access to their medical data.

Our work aims to integrate the different technologies - blockchain for transparent record management, chameleon hashes for maintaining immutability of blockchain and zero knowledge proofs for privacy-preserving verification. This would ensure patients can grant or revoke access through consent-based mechanisms.

CHAPTER-3
PROPOSED SYSTEM

Chapter 3 deals with the system architecture, the functional modules and non functional requirements of the proposed system.

3.1 ARCHITECTURE

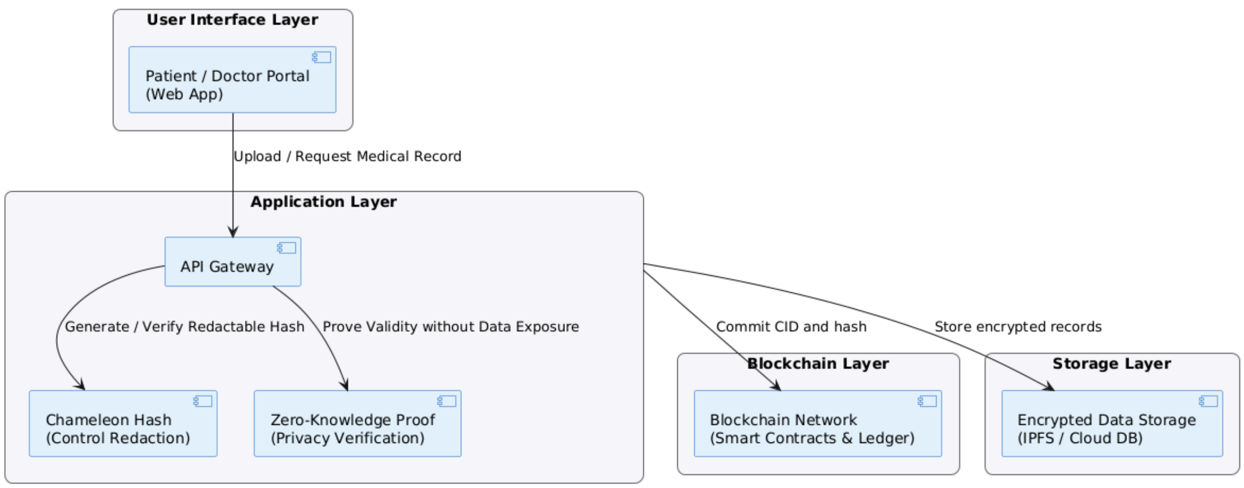


Figure 3.1 High Level Architecture Diagram

3.2 FUNCTIONAL MODULES

3.2.1 Key Generation & Registration

This module provides for the secure creation and verification of a patient's identity in the system. It allows for a unique patient digital identity to be attached to the patient's account, keeping personal credentials secure and private on the device.

3.2.2 Data Encryption & Off-Chain Storage

This module guarantees that patient medical information is encoded securely using standard encryption methodologies, and stored on a private storage network such as IPFS (Interplanetary File System). Each record will be associated with a one-of-a-kind content identifier (CID), which is used to securely access data, if it is necessary to retrieve it.

3.2.3 Data Hashing & On-Chain Registration

This will ensure every patient record is permanently and securely registered on the blockchain. A unique digital fingerprint of the record shifts to the patient's identity in a secure manner. The patient record data may be verified at any time, but once set to the blockchain, it may not be changed, which leads to trust and transparency in the process.

3.2.4 On-Chain Logic & Deployment

This module details the fundamental functions of managing medical records on-chain, including storing new medical records, approving healthcare providers to access records, and verifying data integrity. After development, this module is deployed to a test network to evaluate and validate the technology.

3.2.5 Zero Knowledge Proof Generation

This module allows the patient to demonstrate certain facts about the patient (e.g. age or eligibility) without exposing all of their personally identifiable medical information.

3.2.6 Access Control and Data Retrieval

This module allows an authorized healthcare provider to securely view a patient's EHR. First, the healthcare provider verifies the patient's digitally signed consent and then queries the Smart Contract for the CID and integrity hash associated with that particular record. The encrypted file is then securely retrieved from the IPFS.

3.3 NON FUNCTIONAL REQUIREMENTS

REQUIREMENTS	DESCRIPTION
Performance	The system should efficiently process record uploads, encryption, and verification with minimal delay.
Usability	The interface should be intuitive and support multiple languages.
Scalability	The backend should handle 100+ concurrent users efficiently.
Security	All user data must be encrypted and comply with data protection regulations (e.g., GDPR). Access should be securely

REQUIREMENTS	DESCRIPTION
	managed through authentication and consent control.
Reliability	The system should have 99.9% uptime.

3.4 TECHNOLOGY STACK

CATEGORY	REQUIREMENTS
Backend & Blockchain Tools	Hardhat Solidity Compiler Web3.js / Ethers.js
Cryptography & Proof Tools	ZoKrates Toolkit PyCryptodome
Data Storage Tools	IPFS (Private Cluster)
Programming Languages & Frameworks	Python JavaScript (Node.js)
DApp Development	React.js
Version Control	Github

CHAPTER-4
SYSTEM REQUIREMENTS

Chapter 4 deals with the basic requirements that are expected from the user for smooth functioning of the system.

4.1 HARDWARE REQUIREMENTS

CATEGORY	SPECIFICATION
Processor	Intel Core i5 / AMD Ryzen 5 or higher
RAM	Minimum 8 GB(recommended 16 GB for running Ganache and Hardhat smoothly
Storage	100 GB free disk space
GPU (optional)	Integrated GPU (No dedicated GPU required)
Network	Stable internet connection for blockchain node setup and package installations

4.2 SOFTWARE REQUIREMENTS

CATEGORY	SPECIFICATION
Operating System	Windows 10 / 11 (64-bit) or Ubuntu 22.04 LTS, macOS Monterey (12.0) or later
Blockchain Platform	Ethereum (Local Network using Ganache)
Development Framework	Hardhat v2.26.3
Node.js Environment	Node.js v18 or above, npm package manager
Supporting Tools	IPFS (for off-chain storage), ZoKrates toolkit (for ZKP generation)
Libraries Used	ethers.js, dotenv, @nomicfoundation/hardhat-toolbox

CHAPTER-5
PROJECT WORKFLOW

Chapter 5 details the workflow of the proposed system, explaining the interaction between various components involved in the process.

5.1 SYSTEM WORKFLOW

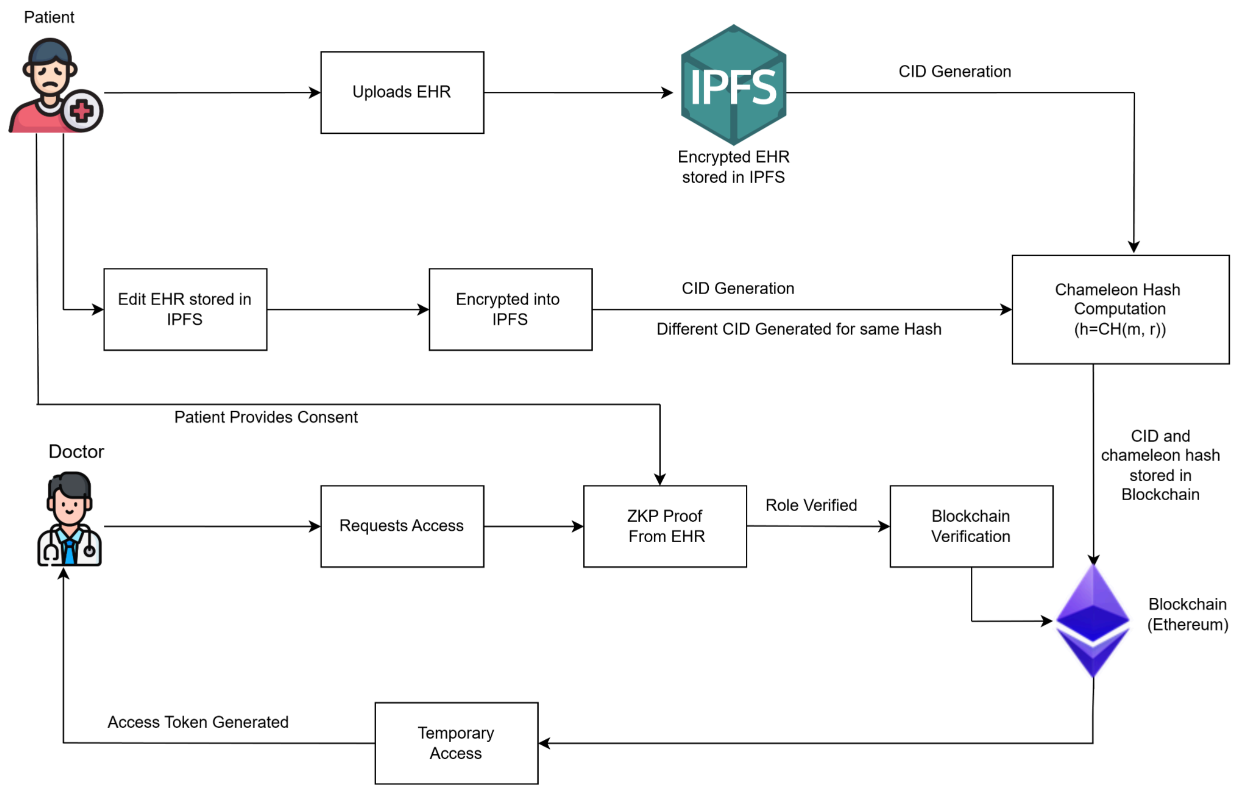


Figure 5.1 System Workflow

CHAPTER-6
SOFTWARE DESIGN

Chapter 6 gives a deep insight on the system through use case diagram and sequence diagram.

6.1 USECASE DIAGRAM

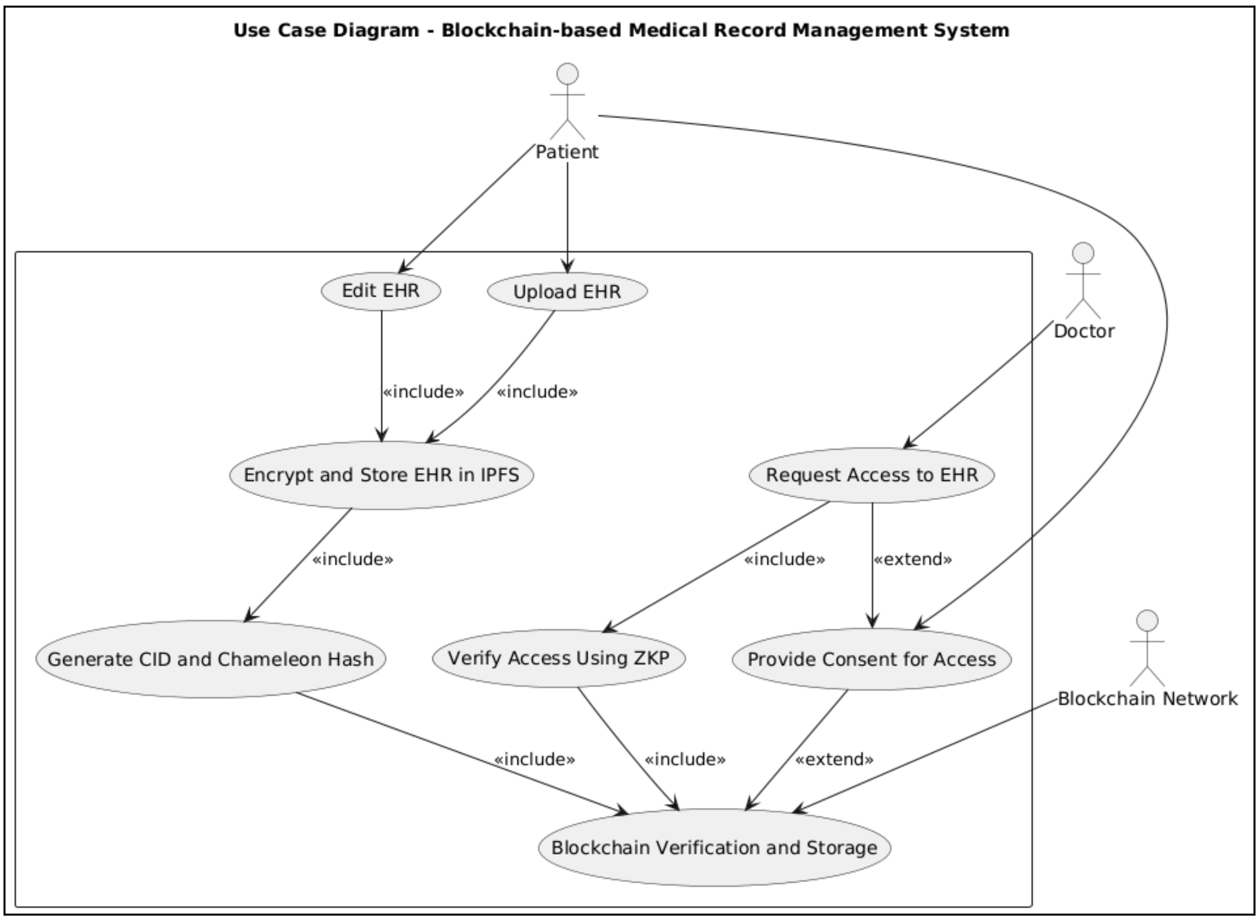


Figure 6.1 Use Case Diagram

6.2 SEQUENCE DIAGRAM

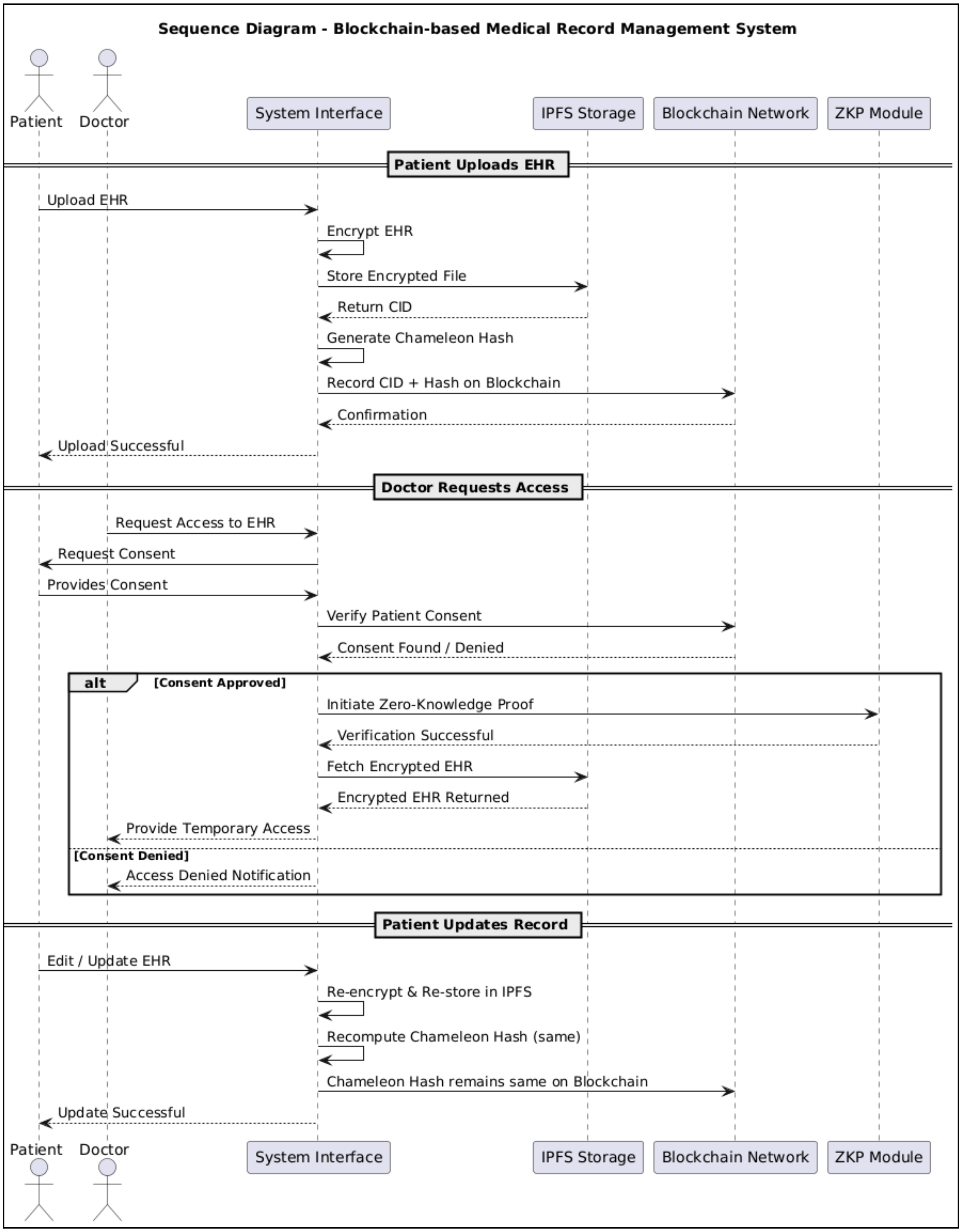


Figure 6.2 Sequence Diagram

CHAPTER-7
IMPLEMENTATION

Chapter 7 explains how the system is implemented with different modules.

7.1 KEY GENERATION

The key generation functionality was implemented to establish a unique cryptographic identity for each user at the time of registration. The process for registration involves generating both a private key and a public key.

Private Key (sk): This key is safely located in the user's local environment where it cannot be accessed by outside parties.

Public Key (pk): This key is passed to the blockchain and interacts with verification and smart contracts.

7.1.1 Comparison between ECC and RSA

ECC and RSA are the two most commonly used algorithms for key generation. ECC stands for Elliptic Curve Cryptography, which uses the mathematical properties of elliptic curves over finite fields to provide security. RSA is an asymmetric cryptographic algorithm based on the mathematical difficulty of factoring large prime numbers.

To analyze the efficiency of both algorithms in key generation, different key sizes in RSA and different curves in ECC were evaluated over their generation time. The below code snippet shows the comparison made.

```
if __name__ == "__main__":
    rsa_key_sizes = [2048, 3072, 4096]
    ecc_curves = ["SECP256R1", "SECP384R1", "SECP521R1", "SECP256K1", "Curve25519", "Curve448"]

    run_key_generation_comparison(rsa_key_sizes, ecc_curves)
```

The below table summarizes the results obtained from the comparison. The equivalent security (bits) were taken from the NIST report.

In this table, it is clear that the RSA algorithm takes longer time for key generation as well as it uses a larger key size to provide the same level of security as in ECC. Within ECC, we chose the SECP256K1 curve because it is the commonly used curve with blockchain.

Algorithm	Key Size (bits)	Key Size (byte	Generation Time (s)	Equivalent Security (bits)
RSA	2048	256	0.03759638309	112
RSA	3072	384	0.127171669	128
RSA	4096	512	0.3963427401	128
SECP256R1 ECC	256	32	0	128
SECP384R1 ECC	384	48	0.0009015536308	192
SECP521R1 ECC	521	65	0.002006120682	256
SECP256K1 ECC	256	32	0.0004061937332	128

7.1.2 ECC Implementation

The **secp256k1** elliptic curve, which is also implemented in Ethereum and Bitcoin networks, was selected to ensure compatibility and high security. The generation process of each key pair followed these steps:

1. **Private Key Generation:**
   A random integer d was chosen within the interval [1, n−1], where n denotes the order of the base point G on the elliptic curve.
2. **Public Key Derivation:**
   The corresponding public key Q was computed using the formula
   $Q=d\times G$

   where × represents scalar multiplication on the elliptic curve. This operation is straightforward to perform but computationally infeasible to reverse due to the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**, which forms the basis of ECC's security.

```python
sk = SigningKey.generate(curve=SECP256k1)
vk = sk.get_verifying_key()
t1 = time.perf_counter()
priv_bytes = sk.to_string()           # 32 bytes
pub_bytes = b"\x04" + vk.to_string()  # uncompressed 65 bytes

# Sign and verify to sanity-check
sig_t0 = time.perf_counter()
sig = sk.sign(msg)
sig_t1 = time.perf_counter()

ver_t0 = time.perf_counter()
ok = vk.verify(sig, msg)
ver_t1 = time.perf_counter()
```

```
(venv) priyadharshinim@Priyadharshinis-MacBook-Air zokrates-work % python keygen.py
{
  "priv_hex": "f59f87893898f9405c8b7b5186480868a0b30eb3b7e11ddf2d5cf5b899962ea7",
  "pub_hex": "0478bdc8b06df4e089a6c85eb4a1e6e0d9687209b92fea7e12cc96f6a51eef72ac67412e01870a7b9129d0d019624db77f8b82aa91c3268de8def9243b561658ed",
  "priv_bytes": 32,
  "pub_bytes": 65,
  "sign_ms": 0.9601249985280447,
  "verify_ms": 3.3561250020284206,
  "verify_ok": true
}
Wrote out/privkey.hex, out/pubkey.hex, and out/ecc_once.json
```

7.1.3 Integration of Pedersen Commitment

To reinforce privacy and data integrity, the **Pedersen Commitment** scheme was integrated into the key-management workflow. A Pedersen commitment locks a secret value x with a random blinding factor r as follows:

$$C = x \cdot G + r \cdot H$$

Where G is the base generator of secp256k1, H is a secondary generator derived from hashing a unique label, C is the commitment point representing the combined value.

This formulation guarantees:

Hiding Property: The original data x cannot be inferred from C since r randomizes the output.

Binding Property: No participant can alter x after commitment, because any change would produce a new C point.

Within the EHR system, this mechanism allows sensitive attributes—such as diagnosis codes or identity tokens—to remain confidential while still enabling later verification through public parameters.

```python
def pedersen_commit(x: int, r: int):
    h_scalar = hash_to_scalar(b"Pedersen H generator")
    H = h_scalar * G
    C = x * G + r * H
    return C

def generate_once():
    x = int.from_bytes(secrets.token_bytes(32), "big") % order
    r = int.from_bytes(secrets.token_bytes(32), "big") % order

    t0 = time.perf_counter()
    C = pedersen_commit(x, r)
    t1 = time.perf_counter()

    out = {
        "x_hex": hex(x)[2:],
        "r_hex": hex(r)[2:],
        "commit_hex": point_to_uncompressed_hex(C),
        "commit_bytes": len(bytes.fromhex(point_to_uncompressed_hex(C))),
        "commit_ms": (t1 - t0) * 1000.0
    }
    return out
```

```
(venv) priyadharshinim@Priyadharshinis-MacBook-Air zokrates-work % python pedersen_py.py

{
  "x_hex": "91c6a91e52018d8c2927a997465f17bdbb771d4a6eac4ffb2abca126db389888",
  "r_hex": "1936405ed1f01c8c0b41bf876fe167853650bb2a15581067b4f63ac27d99ab76",
  "commit_hex": "0419a30fdfaab0ec3805550c19fe3af03117e200bada78363ad4990ddb33a9585178196c2fb07eb4a3c2f0f971f59c119255fd8b88d559adfe1fdcf28910733048",
  "commit_bytes": 65,
  "commit_ms": 13.605749998532701
}
Wrote out/pedersen_once.json
```

7.2 IPFS WITH ENCRYPTION

The process involved encrypting the health records first and then storing them in a private IPFS network. This ensured that even if someone accessed the storage network, the files could not be read without the correct decryption key. A unique identifier was also created for each file, which helped in tracking and retrieving it when required.

7.2.1 Comparison between encryption algorithms

The most common encryption algorithms for PDFs were chosen, namely AES 256 (CBC), AES 256 (GCM), RC4 and compared with different metrics.

Algorithm	Original Size (bytes)	Encrypted Size (bytes)	Encryption Time (s)	Decryption Time (s)	Cipher Entropy (bits)	Byte Difference	Chi-Square
AES-256-CBC	2455504	2455536	0.0082	0.0257	7.9999	2445903.3	0.5988
AES-256-GCM	2455504	2455532	0.006	0.0183	7.9999	2445917.1	0.9771
RC4	2455504	2455504	0.0087	0.0224	7.9999	2445922	0.4506

The above table shows the results of the comparison. The AES 256 (GCM) was chosen for implementation because it had a greater Chi-Square Value than the other two algorithms.

7.2.2 Working

```
def encrypt_folder(input_folder, output_file, password):
    with pyzipper.AESZipFile(output_file, 'w', compression=pyzipper.ZIP_LZMA) as zf:
        zf.setpassword(password.encode('utf-8'))
        zf.setencryption(pyzipper.WZ_AES, nbits=256)

        for foldername, subfolders, filenames in os.walk(input_folder):
            for filename in filenames:
                file_path = os.path.join(foldername, filename)
                arcname = os.path.relpath(file_path, input_folder)
                zf.write(file_path, arcname=arcname)
if __name__ == "__main__":
    input_folder = 'D:\secrete'
    output_file = 'encrypted_folder.zip'
```

The data was first encrypted using the AES-256 algorithm. After encryption, it was compressed into a single ZIP file and uploaded to the IPFS network. A Content Identifier (CID) was automatically generated for every file, serving as a unique address to locate it. If the file was modified, a new CID was generated, which made it easy to detect any tampering. A private IPFS network was set up using the swarm.key file, allowing only trusted nodes to connect and share data securely.

7.2.3 Sample Output

7.2.3.1 Terminal Output (Encryption & Upload)

During this process, the output was taken from the terminal while the file was being encrypted and uploaded.

```
C:\Windows\System32>ipfs add D:\ehr_project\encrypted_folder.zip
 684.16 KiB / 684.16 KiB [==========================================================================] 100.00%
added QmPzihD2JxEykGtzkZRaTdvqAEJSTcMFDMTsW4vZ71z6oX encrypted_folder.zip
 684.16 KiB / 684.16 KiB [==========================================================================] 100.00%
```

7.2.3.2 IPFS Output (Generated CID)

After the upload, an output was shown in the IPFS terminal where a unique CID was created for the encrypted file. This CID was used to find and download the same file later.

Name ↑	Pin Status	Size
encrypted_folder.zip QmPzihD2JxEykGtzkZRaTdvqAEJSTcMFDMTsW4vZ71z6oX		684 KiB
encryptedBD_folder.zip QmbsmnDKCiSGABLykQcUVUp7aGwAoGRWqgd93FkVJ6UGYD		4 MiB

7.3 CHAMELEON HASH

We have successfully generated a chameleon hash tailored for controlled data mutability. The computation involves using a content identifier (CID), a PLONK succinct proof (π), and a randomness value (r) to derive the final hash.

```
(venv) home@Nikethnas-MacBook-Air Chameleon_Hash % python ch_secp256k1.py
Chameleon Hash:
Generated CH keypair:
CID: QmRp3yfy1RyCAHtAP8qvEpnEcPek4xjjE2P8MsWecGn9hn
Chameleon Hash: 4230894469fc9dc973a2096b75060d9fdac0f29f0830b959f3dd577d770a42ad
Computation Time: 0.0009s
Using randomness: 0x397e77fa9076443f36

Testing Collision Finding:
Original (active):  4230894469fc9dc973a2096b75060d9fdac0f29f0830b959f3dd577d770a42ad
Collision (inactive): 4230894469fc9dc973a2096b75060d9fdac0f29f0830b959f3dd577d770a42ad
Hashes match: True
Collision time: 0.0001s

PROPER chameleon hash complete!
```

Our implementation leverages elliptic curve cryptography (secp256k1) to achieve efficient and secure computation using the following function consisting of scalar multiplication and public key aggregation.

```python
def ch_hash(m_bytes: bytes, r: int, pk_pubbytes: bytes):
    # 1) H(m) as scalar
    hm = _hash_to_scalar(m_bytes)
    # 2) r*G -> create PublicKey from secret r (r*G)
    rG = PublicKey.from_valid_secret(r.to_bytes(32, "big"))
    # 3) hm * P => multiply public key P by scalar hm
    P = PublicKey(pk_pubbytes)
    hmP = P.multiply(hm.to_bytes(32, "big"))
    # 4) add points rG + hmP
    # combine_keys can add arbitrary public keys
    S = PublicKey.combine_keys([rG, hmP])
    comp = S.format(compressed=True)  # 33 bytes
    digest = _keccak256(comp).hex()
    return digest, comp
```

We used an ECDSA-based chameleon hash scheme that achieved 0.8 ms computation time for hash generation and 0.1 ms for collision creation through the trapdoor property. This level of performance makes it ideal for high-throughput healthcare systems and on-chain anchoring of records.

The trapdoor property of the ECDSA scheme allows us to support privacy-preserving consent updates where the authorized parties holding the secret trapdoor can produce distinct document states that map to the same on-chain Chameleon Hash. This will help to achieve collision equivalence without altering the blockchain anchor. This capability was validated through working collision experiments that preserved verifiability while enabling sanctioned mutability of sensitive EHR records.

The result is a practical and Ethereum-compatible design that achieves a crucial balance between blockchain immutability and real-world requirements for controlled mutability and patient privacy in electronic health record systems.

7.4 ETHEREUM WITH GANACHE AND HARDHAT

Ethereum was chosen as the blockchain platform because it supports smart contracts and integrates well with Hardhat and Ganache.

7.4.1 Install Prerequisites

Command:

- npm init -y
- npm install --save-dev hardhat@2.26.3 @nomicfoundation/hardhat-toolbox ethers

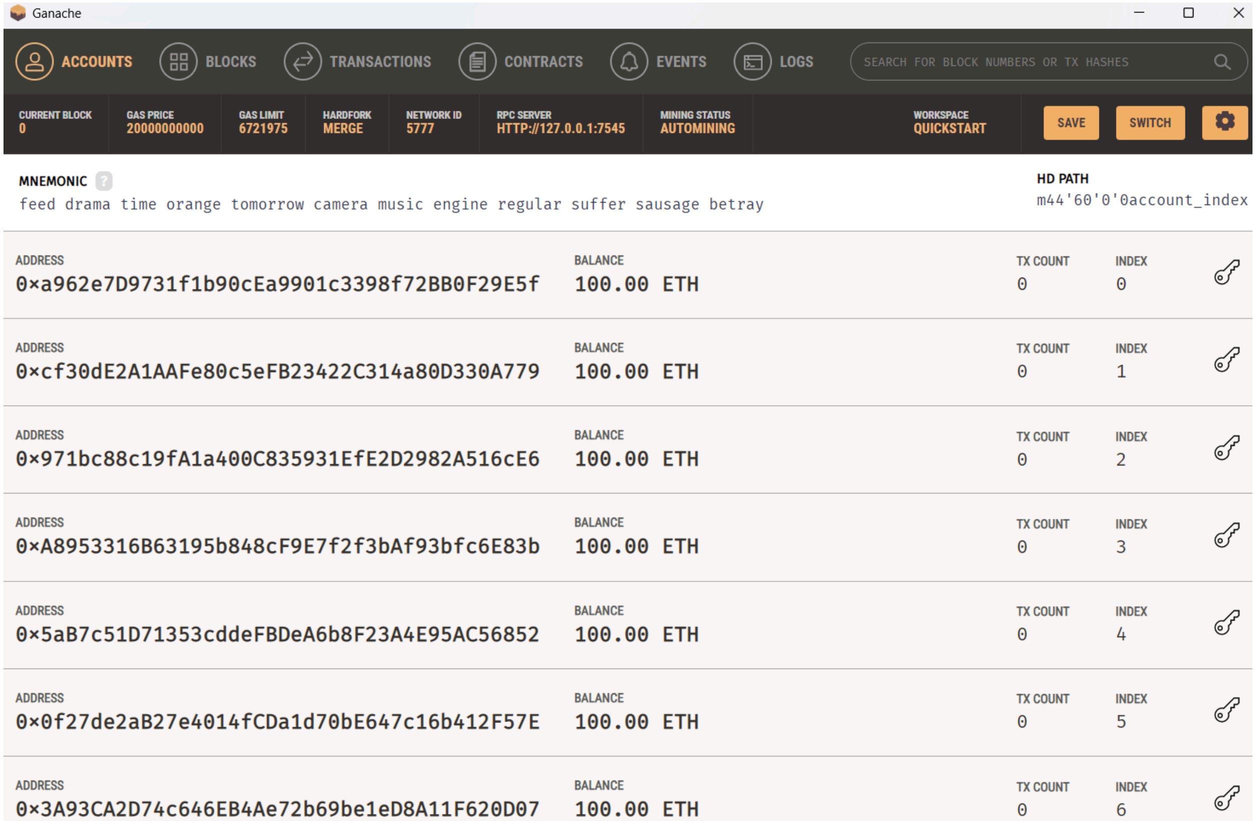
7.4.2 Initialize the Hardhat Project

Command:

- npx hardhat

Then, configure the hardhat.config.js file to include the Ganache network details.This configuration allows Hardhat to connect and deploy contracts to the local blockchain.

7.4.3 Start Ganache



Ganache was launched to create a local Ethereum blockchain for testing the smart contract.Accounts Tab was opened to view the list of automatically generated accounts with test ETH balances.One of these accounts was used to deploy the smart contract from Hardhat.After successful deployment, the Contracts Tab in Ganache displayed the newly deployed contract along with its address.The Transactions Tab showed the transaction details confirming the deployment.This verified that the local blockchain and Hardhat were properly connected.

7.4.4 Create smart contract

Self executing digital agreement with the terms of the contract written directly into lines of code, stored and run on blockchain network.In this project, the smart contract records:

- The hash value generated by the Chameleon Hash module.
- The encrypted CID from the IPFS module.

KeyRegistry.sol M

hardhat > key-registry-hardhat > contracts > KeyRegistry.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract KeyRegistry {
    struct Record {
        address owner;
        bytes pubKey;
        bytes32 h;
        string encryptedCid;
        uint256 timestamp;
        bool exists;
    }
    mapping(address => Record) private records;
    // Register public key
    function registerKey(bytes calldata pubKey) external {
        require(!records[msg.sender].exists, "Already registered");
        records[msg.sender] = Record({
            owner: msg.sender,
            pubKey: pubKey,
            h: bytes32(0),
            encryptedCid: "",
            timestamp: block.timestamp,
            exists: true
        });
    }
    // Store hash and encrypted CID
    function storeData(bytes32 h, string calldata encryptedCid) external {
        require(records[msg.sender].exists, "Key not registered");
        records[msg.sender].h = h;
        records[msg.sender].encryptedCid = encryptedCid;
        records[msg.sender].timestamp = block.timestamp;
    }
    // Retrieve record
    function getRecord(address owner) external view returns (bytes memory, bytes32, string memory, uint256, bool) {
        Record storage r = records[owner];
        return (r.pubKey, r.h, r.encryptedCid, r.timestamp, r.exists);
    }
    // Verify hash
    function verifyHash(bytes32 hCandidate) external view returns (bool) {
        require(records[msg.sender].exists, "Key not registered");
        return records[msg.sender].h == hCandidate;
    }
}
```

7.4.5 Compile the Smart Contract

Command:

- npx hardhat compile

7.4.6 Deploy the Smart Contract on Local Ganache Blockchain

Command:

- npx hardhat run scripts/deploy.js --network ganache

After deployment, the terminal displays the contract address.The deployed contract can also be viewed in Ganache → Contracts tab.

7.5 ZERO KNOWLEDGE PROOF GENERATION

to be filled

7.6 D-APP DEVELOPMENT

to be filled

7.7 UI SCREENSHOTS

#to be filled

7.8 PERFORMANCE METRICS

to be filled

CHAPTER-8
CONCLUSION

Chapter 8 explains how we plan to expand the system with the use of Machine Learning

8.1 FUTURE WORK

The project is planned to extend with the addition of federated learning or swarm learning to provide integration between hospitals in a decentralized way. Currently the system is complete for a single hospital. The main goals of the phase II is to identify a use case within this project to integrate machine learning, and to implement it in a federated or swarm pattern, or maybe in both ways to find which is advantageous.

8.2 BIBLIOGRAPHY

[1] H. S. A. Fang, T. H. Tan, Y. F. C. Tan, and C. J. M. Tan, “Blockchain personal health records: A systematic review,” *Journal of Medical Internet Research*, vol. 23, no. 4, p. 25094, 2021.

[2] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989.

[3] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge,” *Aztec & Pi Squared*, 2025.

[4] V. Shoup, “A composition theorem for universal one-way hash functions,” *Cryptology ePrint Archive*, Report 1999/017, 1999.

[5] S. Samudrala *et al.*, “Performance analysis of zero-knowledge proofs,” in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, Vancouver, BC, Canada, 2024, pp. 144–155. doi: 10.1109/IISWC63097.2024.00022