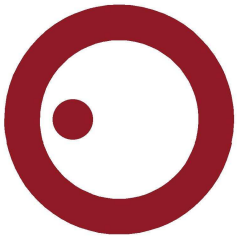


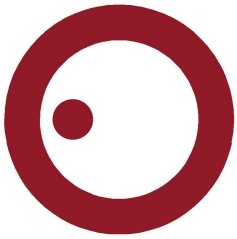
escola superior de tecnologia e gestão
instituto politécnico de leiria

Polytechnic University of Leiria
School of Technology and Management
Department of Computer Engineering
Master's Degree in Computer Engineering - Mobile Computing

DIGITAL HEALTH IDENTITY WITH BLOCKCHAIN

FREDERICO PINTO CARVALHO

Leiria, september of 2024



escola superior de tecnologia e gestão
instituto politécnico de leiria

Polytechnic University of Leiria
School of Technology and Management
Department of Computer Engineering
Master's Degree in Computer Engineering - Mobile Computing

DIGITAL HEALTH IDENTITY WITH BLOCKCHAIN

FREDERICO PINTO CARVALHO
Number: 2220656

Project Report under the supervision of Professor Doctor Marisa da Silva Maximiano (marisa.maximiano@ipleiria.pt) and Professor Ricardo Jorge Pereira Gomes (ricardo.p.gomes@ipleiria.pt)

Leiria, september of 2024

ACKNOWLEDGEMENTS

I want to express my sincere gratitude to the people and organizations that played a key role in the completion of this thesis. First and foremost, I owe an immeasurable debt of gratitude to my professors and the coordinators of this project, whose limitless patience and insightful feedback guided me through this academic journey. To my classmates and groupmates, in particular, to my dear friends, I am grateful for their unwavering moral support. I must acknowledge the unfailing support of my family, with a special mention to my girlfriend. Their unwavering belief in me has been the keystone of my perseverance and motivation throughout this process. Finally, a note of gratitude to my feline companion, who is always fun and emotionally supportive, and who has provided moments of comfort amid academic challenges.

ABSTRACT

The increasing demand for innovative healthcare systems and efficient information management has pushed institutions and private consortia to explore decentralized solutions to safeguard patients' sensitive information. These solutions aim to enhance privacy and allow controlled access to specific entities requesting patient data. One of the key technologies explored to address these challenges is blockchain, which offers a decentralized, secure, and user-centric approach to managing digital health information.

The major problem this work addresses, revolves around two critical aspects: identity management and access management. In traditional healthcare systems, the risk of identity exploits, personification, and linking patient data to identifiable individuals through mining algorithms poses a significant threat to patient privacy. Data mining techniques can aggregate and correlate health records with specific patients, even without direct identifiers, leading to breaches in privacy and unauthorized disclosure of patient information. The goal of this work is to develop a system that prevents such identity exploitation, ensuring that the identity behind the patient cannot be discovered nor connected to their health data.

On the access management side, the challenge is ensuring that patients have full control over their health data. The system must allow patients to securely grant or revoke access to their data, while ensuring that even sophisticated mining algorithms cannot infer the ownership or identity behind the data, nor decrypt it without explicit authorization. This ensures that patient data remains private and secure, accessible only by authorized healthcare providers, and that no unauthorized third party can exploit the data for research or other purposes without consent.

The objective of this work is to develop a decentralized and privacy-preserving system for managing patient data, enabling secure sharing of information between patients and healthcare providers. This system leverages blockchain technology, smart contracts, and Zero-Knowledge Proofs to ensure that patients maintain full control over their data while preventing unauthorized data aggregation, analysis, or decryption.

To achieve these goals, this work involved the study of existing blockchain-based solutions, evaluating their feasibility for healthcare data management. Three

potential approaches were considered for managing identity and resources. After thorough analysis, Zero-Knowledge Proofs were selected as the most suitable solution due to their ability to provide privacy-preserving mechanisms without revealing sensitive information. The implementation of the solution was followed by rigorous testing and validation to ensure its functional accuracy and alignment with privacy objectives.

The results of this work demonstrate a proof-of-concept implementation that successfully integrates blockchain technology with Zero-Knowledge Proofs, allowing secure, patient-controlled data access while protecting against unauthorized data mining and ensuring the privacy of the patient's identity. Central to this solution is the development of a "Data Sharing Agreement," which addresses the access management issue by enabling patients to selectively grant healthcare providers access to their medical data. Further studies are encouraged to integrate this solution into real-world healthcare systems, addressing scalability and operational challenges.

Keywords: blockchain; privacy; data sharing agreements; zero-knowledge proofs; electronic medical records

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

AA	Account Abstraction.
ABI	Application Binary Interface.
API	Application Programming Interface.
CPR	Computerized Patient Record.
DMR	Digital Medical Records.
DOM	Document Object Model.
DRM	Digital Rights Management.
EBSI	European Blockchain Services Infrastructure.
ECC	Elliptic Curve Cryptography.
EHR	Eletronic Health Records.
EMR	Eletronic Medical Records.
EOA	Externally Owned Account.
ERC	Ethereum Request for Comment.
ETH	Ethereum currency.
EVM	Ethereum Virtual Machine.
FHIR	Fast Healthcare Interoperability Resources.
GCP	Google Cloud Platform.
HIPAA	Health Insurance Portability and Accountability Act.
HIS	Health Information Systems.
HL7	Health Level Seven.

JSON	JavaScript Object Notation.
JSX	JavaScript XML.
KPI	Key Performance Indicator.
LOINC	Logical Observation Identifiers Names and Codes.
ONC	National Coordinator for Health Information Technology's.
PBFT	Practical Byzantine Fault Tolerance.
PHR	Personal Health Records.
PoB	Proof of Burn.
PoC	Proof of Concept.
PoS	Proof of Stake.
PoW	Proof of Work.
RPC	Remote Procedure Call.
RSA	Rivest-Shamir-Adleman.
SDK	Software Development Kit.
SNOMED	Systematized Nomenclature of Medicine—Clinical Terms.
TPS	Transactions Per Second.
UI	User Interface.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
UX	User Experience.

ZKP	Zero-Knowledge Proof.
zkSNARK	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge.

1

INTRODUCTION

The growing complexity of healthcare systems and the need for effective information management are driving both institutions and private entities towards adopting decentralized solutions. In this context, Blockchain technology has emerged as a key enabler of decentralization, offering a secure, transparent, and distributed way to manage sensitive data. These Blockchain-based approaches prioritize patient data security and empower individuals to selectively grant or revoke access to their information while ensuring that no central authority holds complete power over it.

A critical problem in healthcare information management is the need for robust and secure access management to medical records. In decentralized systems, medical records are typically provided either by participating healthcare providers or by the patients themselves. Each medical record in such a system must have a well defined set of authorizations to control who can access the data. For example, the patient, as the data owner, should always have permanent access to their medical records. Furthermore, there must be provisions for granting temporary access to specific medical professionals or healthcare providers for a fixed period, ensuring that access rights are clearly defined and limited over time. These authorizations must be securely stored on the blockchain to maintain an immutable and transparent record of all access permissions. To facilitate secure access, the system must support efficient queries that allow participants to determine which medical records they have permission to access. The patient's wallet, which acts as their digital interface, should be able to list all medical records accessible to the patient and provide a mechanism for viewing these records. This approach ensures that patients maintain control over their data, aligning with the overarching goal of data privacy and patient empowerment.

The identity management requirements of a decentralized healthcare platform introduces additional complexities. Each patient is represented by a unique digital wallet, which is associated with a single blockchain account or identifier. Similarly, each healthcare provider must have a unique blockchain ID. The system must accommodate multiple relationships, where a single patient wallet can have relationships with multiple healthcare providers, and conversely, a healthcare provider can interact

with multiple patient wallets. Healthcare providers typically maintain several medical information repositories that contain patient data and complementary diagnostic methods (medical exams) results. Each piece of information or complementary diagnostic methods (medical exams) results stored in these repositories must be linked to a unique identifier on the blockchain, which references both the healthcare provider and the patient. Additionally, healthcare providers often employ multiple health professionals, further increasing the complexity managing access rights and relationships.

Given these identity management and access control requirements, the proposed system must provide a simplified mechanism for managing identities and relationships while ensuring the privacy and security of all participants. Privacy protection is paramount: the system should avoid using a single identifier that could facilitate de-anonymization of patient data. It should also ensure that only participants in a direct relationship have access to the relevant data, effectively preventing unauthorized access and potential data breaches.

Several research efforts have explored solutions to these challenges, with a particular focus on using cryptographic techniques, such as Zero-Knowledge Proofs [1], to enhance privacy and security. These methods can help mask the relationships between patients, healthcare professionals, and Healthcare Providers, making it difficult for third parties to infer sensitive information from the blockchain data. Furthermore, anonymization techniques are often employed to prevent data mining algorithms from linking medical records to specific individuals or healthcare entities, thus preserving confidentiality in scenarios involving clinical research or data sharing [2].

Overall, the system must balance the competing demands of data accessibility, privacy protection, and efficient management of relationships between patients, healthcare providers, and healthcare professionals. By integrating advanced cryptographic methods and decentralized identity management protocols [3], the platform aims to provide a secure and privacy-preserving solution that empowers patients while maintaining the integrity and confidentiality of their medical data.

Our approach addresses privacy concerns from two critical perspectives: first, by proposing a method to ensure the confidentiality of individual patient data, and second, by introducing safeguards against potential data mining attempts on public blockchain data. By leveraging advanced cryptographic techniques, including Zero-Knowledge Proofs (ZKP), we aim to mask the relationships between patients, healthcare professionals, and healthcare providers, thereby enhancing privacy protec-

tion in medical data sharing. The followed methodology was centered on practically analyzing the viability of using blockchain technology to manage relationships between healthcare professionals and patients while maintaining privacy.

Moreover, to mitigate the risk of data mining and unauthorized access, it is essential to obscure information that could otherwise be accumulated and linked by data mining algorithms. This is particularly important in contexts where third parties access patient data without explicit consent and potentially use it for purposes such as clinical research. The development and implementation of effective anonymization methods within blockchain-based healthcare systems represent a key area for innovation, with the potential to reshape healthcare data management by better balancing the need for data utility and patient privacy in an increasingly digital healthcare environment [4], [5].

This work forms part of a broader initiative, the BlockchainPT – Decentralize Portugal with Blockchain Agenda, which explores the potential of blockchain technology within the Portuguese economic landscape. This initiative involves several work packages, with the present work being part of Work Package 2 – Health and Wellbeing [1]. Within this framework, the Bio Global Health Platform (BioGHP) aims to design and implement a platform that enhances the user experience for both healthcare professionals and patients in managing medical data and associated processes, such as data sharing agreements. The ultimate objective is to develop a privacy-preserving, decentralized health record management system.

This document is organized as follows: Chapter 2 covers key concepts related to the technologies proposed to address privacy and data-sharing challenges. Chapter 3 reviews relevant state-of-the-art research and developments in this area. Chapter 4 introduces and discusses the proposed architecture for privacy-preserving data management in healthcare. Finally, Chapter 5 presents the conclusions and outlines future research directions.

1 https://blockchain.void.pt/

2

BACKGROUND

This chapter presents the foundational concepts that underpin the development of a decentralized, privacy-preserving medical data-sharing platform. These concepts are central to understanding the challenges and technological solutions proposed in this work. Additionally, we review relevant research efforts in the field, highlighting how other investigators are addressing similar problems, particularly the privacy and data security concerns inherent in decentralized healthcare systems.

In effort to explore the potential and implementation of blockchain technology within the healthcare sector, it is imperative to first lay the groundwork by elucidating a series of fundamental and advanced concepts that underpin this technology and its applications. This section serves as a preparatory platform, providing a structured overview of key terminologies that are essential for a comprehensive understanding of the subsequent discussions and analyses.

The primary aim of this section is to equip readers with a thorough understanding of the critical technological frameworks and methodologies that will be referenced throughout this document. This knowledge is vital for appreciating the depth and breadth of blockchain technology's capabilities, limitations, and innovative applications within healthcare. By presenting these concepts in a cohesive and structured manner, the section seeks to facilitate a seamless transition into more complex discussions and the exploration of a proof of concept in subsequent sections.

In the pursuit to explore the feasibility and impact of blockchain technology within the healthcare sector, we initiate our discourse with an introduction to blockchain technology itself. This foundational technology acts as the backbone of our entire discussion, establishing its critical role in the digital transformation of healthcare.

Next, we examine Smart Contracts [6] and Smart Accounts [7], crucial for automating secure and transparent healthcare transactions on the blockchain. Account Abstraction further simplifies interaction with these technologies, enhancing usability for healthcare stakeholders [8]. This streamlined approach not only makes blockchain more accessible but also paves the way for its wider adoption in healthcare, merging the technical robustness of Smart Contracts with user-centric interfaces.

Subsequently, we explore Zero-Knowledge Proofs (ZKP) [1], an advanced cryptographic principle that enforces privacy-preserving transactions. In the context of healthcare, where data sensitivity is paramount, this concept is instrumental in facilitating secure and confidential handling of health data.

Finally, we contextualize the application of blockchain technology in the management and security of Electronic Medical Records (EMR) [9]. This section highlights how blockchain can serve as a transformative force in redefining the management, accessibility, and security of EMR, marking a significant leap towards improved healthcare delivery and patient outcomes.

2.1 BLOCKCHAIN

Initially found present in the cryptocurrencies ecosystem, Blockchain is a peer-to-peer distributed ledger technology, notable for its decentralized nature [10]. Unlike traditional centralized systems where a single entity controls the data, in a blockchain, the data is distributed across a network of computers, each referred to as a 'node' [11]. This decentralization ensures that no single entity has complete control over the ledger. Each node in the network holds a copy of the entire ledger, and all nodes participate in validating and recording transactions [12]. Transactions are grouped into blocks, and each block is cryptographically linked to the preceding one, forming a chain.

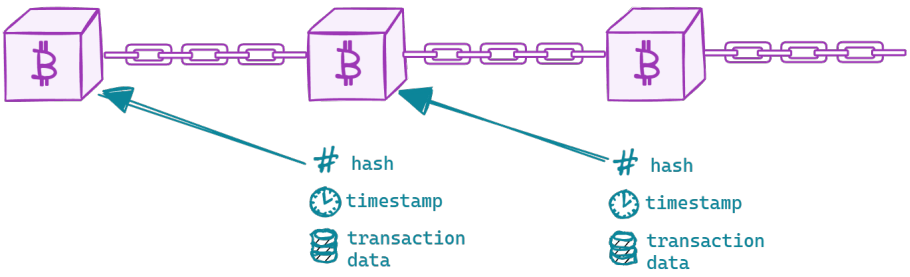


Figure 1: Blockchain Overview

As Figure 1 shows, each block contains a cryptographic hash of the previous block, a timestamp, and transaction data, creating an immutable and transparent record of all transactions [13]. This structure not only ensures data integrity and transparency but also makes the blockchain highly resistant to tampering and revision. Assuming the network enforces its consensus mechanism [14], if a change is made to the data in one block, it would require alterations in all subsequent blocks and the consensus

of the majority of the network, which is practically infeasible, thereby securing the data against unauthorized modifications.

2.1.1 *The Blockchain Concept*

Blockchain can be described as an *humoungous* string with nodes that cannot be directly modified nor removed. Every knot has awareness of its predecessor and if available already, its successor. The main reason why Blockchain exists is the necessity of having an unbreakable system without someone controlling it, thus making the key attribute of blockchain its decentralization. No central authority controls the content added to the blockchain. Instead, the newly added nodes are to the blockchain are agreed upon in a peer-to-peer network using various consensus protocols [14]. Persistency is another core characteristic of blockchain. Like the example mentioned in the begining of this section, it is practically impossible to untie a well-formed node from a string. The same concept is applied here in the blockchain. After an entry is accepted onto the blockchain (mined block [15]) and due to the implementation of the chain as a distributed ledger where all the network nodes maintain a local copy of the chain, there is no available mechanism to delete the newly added entry from all the nodes, thus maintaining the same registry across the nodes. Although it may not be possible to remove entries from the blockchain, there are solutions for updating information on some blockchains. Last but no least, since the information of the blocks is stored in a non-human readable way (encrypted [16]), blockchain achieves another very important key which is the anonymization or pseudo-anonymization. However, this does not mean that data aggregation is not feasible. Auditing and traceability may be possible because blockchain links the newer block to the previous by including the hash of the latter, and in this way forming a chain of blocks. The transactions in the blocks are formed in a Merkle tree [17]. Also known as a hash tree, used for verifying the integrity of large sets of data. Each block contains a Merkle tree, where transactional data is organized into a tree structure. The *leaves* of this tree are hash values [18] of individual transactions. These hash values are then paired, concatenated, and hashed again, continuing this process recursively until a single hash remains, known as the Merkle root [18]. This Merkle root is stored in the block header, serving as a unique fingerprint for all transactions in that block.

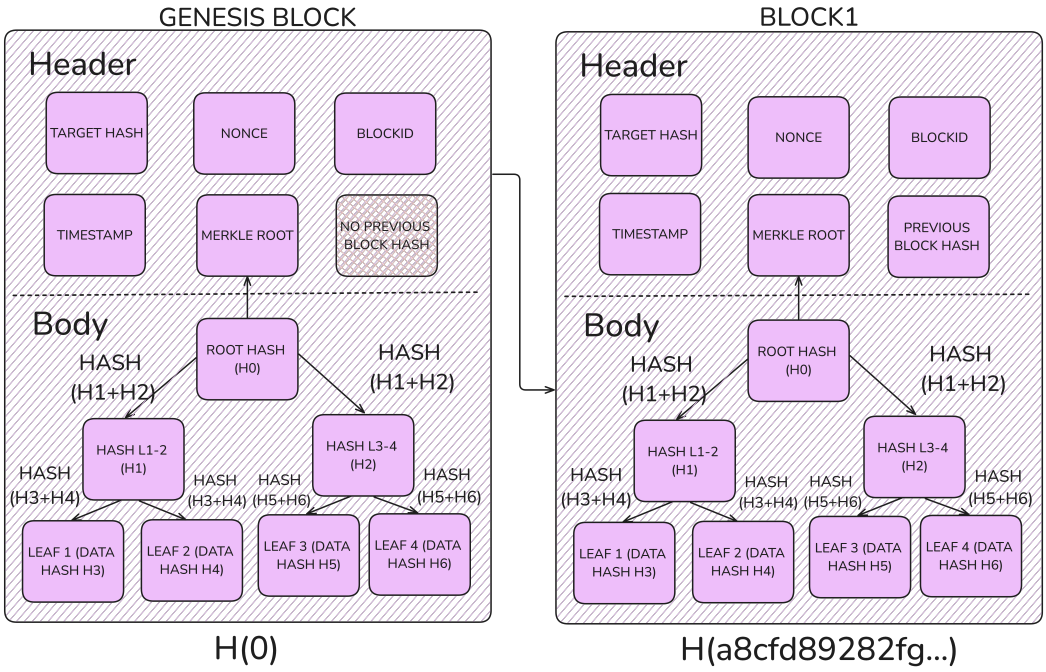


Figure 2: Generic Structure of Blockchain

For a more comprehensive review, Figure 2 presents a generic structure about the chain, where each block is subdivided by: a header and it's body. The block header, consists of metadata, which is made up with the block message, the previously block hash, the Merkle Root, a timestamp and a nonce. Note that particularly the Genesis or First Block has no hash value (0), and there is no previous block hash. Security wise, if a malicious actor wants to corrupt the data (change) of a certain's block header, he must start doing that in the Genesis block. To some extend, this difficulty can discourage malicious tampering of the data. When appending a new block to the chain, a timestamp is stored in the block header of each block to ensure the traceability of the stored data in the blockchain. The body stores the detailed data and the Merkle Tree itself. If the content block in the blockchain has been changed, it will result in corresponding changes to all the subsequent blocks connected to it. Consequently, a totally different blockchain is formed.

2.1.2 *Types of blockchains*

As shown in Figure 3, there are four main types of blockchain implementations: Public (or Permissionless), Private (or Permissioned), Hybrid, and Consortium (Federated). Each of these types will be described in detail in the following sections.

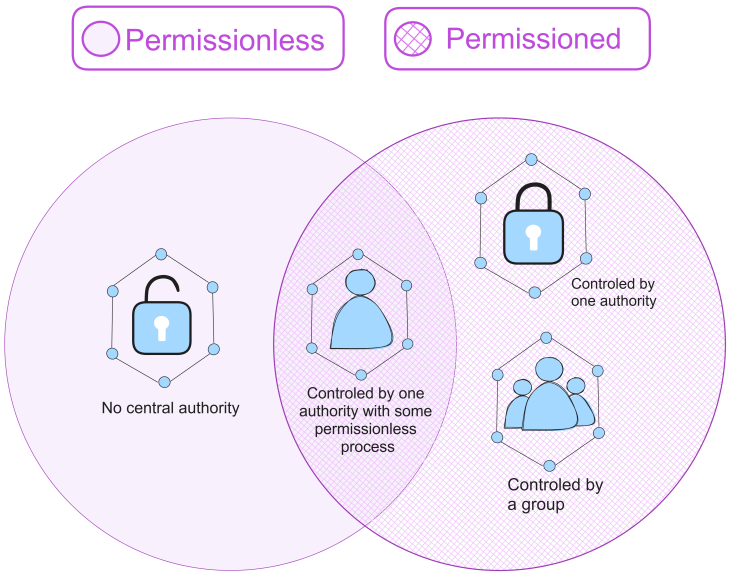


Figure 3: Types of blockchains

2.1.3 *Public or Permissionless blockchains*

The Public (Permissionless) Blockchains (seen in Figure 3) [19] are open to everyone, where any participant can join and contribute to the consensus process and the core software changes. In public blockchains, all data can be viewed by anyone, making them ideal for applications like cryptocurrencies. Bitcoin [10] and Ethereum's [20] main chain are notable examples of public permissionless chains.

The first and most well-known public blockchain, Bitcoin, is primarily used for cryptocurrency transactions. It operates on a Proof-of-Work (PoW) [15] consensus mechanism, emphasizing security and decentralization, but facing scalability and energy consumption challenges [21].

Ethereum is a programmable blockchain allowing any user to execute complex algorithmic code on its Ethereum Virtual Machine (EVM). It supports two types of accounts: Externally Owned Accounts (EOA) controlled by a private key, and Contract Accounts, functioning as autonomous agents governed by smart contracts. Ethereum uses a Proof-of-Work (PoW) consensus mechanism, where users participate in mining to achieve consensus. However, PoW's high energy consumption and limited transaction throughput have led to the exploration of alternatives like Proof-of-Stake (PoS) and Proof-of-Burn (PoB) [15].

2.1.4 *Private or Permissioned blockchains*

In private blockchains, the network is distributed but often centralized, managed by one central authority. Participation is limited to selected nodes. These blockchains are typically used by organizations for internal processes due to their enhanced control and privacy. This document won't focus in this topic as its considered irrelevant for our use case, namely because one of the key requirements of this research is to implement a widely available system that can be used publicly but with constraints. Therefore, private blockchains are not the most well fitted candidate for the healthcare purpose, as it is very coupled and participants who enroll in this blockchain have their identity well known across the ledger and their actions are traceable. The same goes for data aggregation and non-anonymization of each party involved [19].

An example of a permissioned blockhain would be Hyperledger [22], an open-source initiative hosted by the Linux Foundation. Designed to support the development of enterprise grade blockchain-based applications. Unlike public blockchains, which allow any user to participate and validate transactions, Hyperledger operates on a restricted basis, so called, "permissioned", meaning that participants are pre-selected and have specific roles within the network.

With its modular architecture, Hyperledger enables the integration of various consensus mechanisms to suit different application needs. One such mechanism is Practical Byzantine Fault Tolerance (PBFT) [23], which is designed to protect the system against failures by ensuring that the network reaches consensus even in the presence of faulty or malicious nodes [24].

Similarly to other blockchains that implement mechanisms that introduce programmable code that operates upon the ledger itself (e.g: Ethereum's Smart Contracts), Hyperledger also introduces its own concept: Chaincode [24]. It serves two primary purposes: encapsulating the business logic of an application and manage an "associated world state". This world state is a database that records the current state of all ledger data, represented as key-value pairs. This organization facilitates efficient data retrieval and synchronization across the network, ensuring that all participants have consistent and up-to-date information [23].

2.1.5 *Hybrid and Consortium blockchains*

These two remaining types of blockchains are basically a subset of the more abstract categories, private and public. Hybrid blockchains merge both of these in a single implementation, thus taking advantage of having a controlled access, but at the same time publicly verifiability. This concept can be easily described as to each individual (user of the blockchain) having their own private copy of their blockchain, and periodically update a public blockchain with only the data that they desire so. As Marar *et al.* [25] mentions, in order this to work, firstly, the user must be accepted onto the blockchain. Prior to this onboarding, the user now withholds a local copy of his own private blockchain. Whenever a user wants to send a transaction he broadcasts a request to the network. By this time, a miner, someone who is trusted in the network, is elected to do the work and compute the user's information based on his request. This computed data includes a global hash and a private hash that is sent back to the user. It is this private hash that will be used later to link the users transactions (data blocks) together. An example use case for this type of blockchain would be two entities that want to perform a deal but don't want to fully disclosure the contents of the deal, but rather have it publicly incited somewhere, so that it can be easily verified that, indeed, there is a contract or a relationship between these two parties [26], [27].

Consortium blockchains [28], on the other hand, are governed by a group of organizations, rather than a single entity or neither (the public). This collaborative model supports a pre-defined consensus mechanism, e.g: Proof of Stake (PoS) [14], where only members of have a stake on the network can approve transactions, tailored to the specific needs of the consortium, ensuring efficiency and trust among stakeholders. The blockchain's architecture facilitates shared control over the ledger, with permissions and access rights distributed among consortium members, enhancing security and mitigating single points of failure. The interest of implementing a blockchain as a consortium would be applied to entities such as banks or even governments. Whereas every user registered in the network has its identity well know and approved before hand. Each transaction is verified by a very restricted group of members and must be easily proved that member X did Y. This blockchain approach has multiple security features, and reduced transaction times, due to its restricted access. Nevertheless, there are some limitations to be considered, mainly how to coordinate this consortium because there isn't a single entity that has control over the network [28], [29].

For a more comprehensive overview of the different blockchain types discussed, please refer to Table 1.

Table 1: Comparison of Blockchain Types [15], [28], [30]

Metric	**Public**	**Private**	**Consortium**	**Hybrid**
Governance	Distributed	Centralized	Decentralized	Varies
Consensus Participation	Open to all	Authority-defined	Pre-selected nodes	Segment-dependent
Read Permissions	Public	Restricted	Restricted within group	Mixed
Efficiency	Low	High	Moderate	Mixed
Energy Use	Very High	Low	Moderate	Moderate-High
Immutability	Highly immutable	Potentially mutable	Less mutable	Hybrid
Nature	Permissionless	Permissioned	Permissioned	Depends
Network Actors	Anonymous	Known	Preselected	Varies
Speed	Slow	Fast	Moderate	Variable
Security	High	Lower	Moderate	Customizable
Privileges	Open	Admin-defined	Equal	Variable
Examples	Bitcoin, Ethereum	MONAX, Multichain	Hyperledger, Corda	Mixed

2.2 PRIVATE KEYS

A private key in cryptography is a secret number that allows a user to perform cryptographic functions, such as decryption and digital signatures [3]. In the context of blockchain technology and cryptocurrencies, a private key is an essential component that allows users to access and manage their digital assets securely. Private keys are typically represented as a long string of alphanumeric characters, generated by cryptographic algorithms to ensure a high level of security. The strength of a private key lies in its randomness and complexity, which makes it practically impossible for an unauthorized entity to guess or derive it through computational methods. In blockchain systems, a private key is used to sign transactions, providing proof that the transaction has been authorized by the owner of the associated digital assets. The private key is mathematically linked to a public key, which is shared openly and used to verify the signatures created by the private key. The security of digital

assets is thus heavily reliant on the secure storage and management of private keys. If a private key is lost or compromised, the corresponding digital assets can be permanently lost or stolen [31].

2.3 GAS FEES

Transaction fees, also widely known as *Gas Fees*, are required to perform transactions or execute Smart Contracts on the network. Gas fees are a form of incentive for miners, who validate and process transactions by adding them to the blockchain. The amount of gas required for a transaction depends on its complexity and the computational resources needed to execute it. Each operation in a smart contract consumes a certain amount of gas, and more complex operations require more gas. Users specify a gas limit, which is the maximum amount of gas they are willing to spend on a transaction, and a gas price, which is the amount they are willing to pay per unit of gas. Gas fees serve several critical purposes. First, they compensate miners for their work in processing transactions and securing the network. Second, by requiring a fee for each transaction, gas fees help prevent spam and ensure that network resources are used efficiently. Third, gas fees allocate network resources to users who value them the most, and those willing to pay higher fees have their transactions processed more quickly. The dynamic nature of gas fees means they fluctuate based on network demand. During periods of high network activity, gas fees can increase significantly, reflecting the higher competition for processing capacity. Conversely, during low activity periods, gas fees tend to be lower [20].

2.4 SMART ACCOUNTS AND SMART CONTRACTS

In the context of the Ethereum blockchain, Smart Accounts represent a significant advancement in wallet technology, often implementing principles from proposed standards such as ERC-4337 (EIP4337) [7]. These accounts significantly enhance the capabilities of traditional wallets, such as Externally Owned Accounts (EOAs) [8]. While EOAs have limited functionality and lack internal logic or execution capabilities, Smart Accounts expand beyond these constraints. They allow developers to program the account itself, transforming it into a sophisticated entity capable of managing its own state and logic. This evolution markedly increases the account's functionality and versatility within the blockchain ecosystem.

Unlike traditional wallets, a smart account can execute contract code and maintain its own state and management. One of its primary benefits over EOAs is improved key management and user experience. EOAs require owners to store and safeguard their credentials (usually in the form of private keys) and manually approve each transaction. A common example is using a Metamask account[1], where a confirmation pop-up appears in the browser for the user to accept each transaction. Alternatively, a smart account can be programmed to handle many of these operations on behalf of the user. It can manage transaction signing, bundling, and execution using alternative authentication methods, potentially reducing the risk associated with direct private key management [8].

Smart accounts also offer programmable account recovery methods, a significant advantage over traditional EOAs. For instance, a smart account could be programmed to require multiple signatures for high-value transactions, or to implement a time-lock feature for added security. However, it's important to note that while smart accounts offer numerous advantages, they also introduce a level of complexity and potential security considerations that need to be carefully addressed in their implementation and use.

Building upon the concepts introduced in ERC-4337 [7], the more recent ERC-6900 proposal [32] aims to standardize modular smart contract accounts. This standard focuses on creating a framework for composable and upgradeable smart accounts, allowing for greater flexibility and customization. ERC-6900 introduces the concept of plugins - modular components that can be added or removed from a smart account to modify its behavior. This approach enables accounts to evolve over time, adapting to new use cases or security requirements, without the need for users to migrate to entirely new accounts. The modular nature of ERC-6900 aligns well with the healthcare sector's need for adaptable, secure, and customizable data management solutions.

Smart contracts are self-executing programs that run on a blockchain. Unlike traditional contracts, smart contracts are entirely digital and automatically enforce the terms of an agreement without the need for intermediaries. They are written in code (often Solidity for Ethereum [33]) and deployed to the blockchain, where they can interact with accounts, transfer value, and maintain their own state [6].

The key feature of smart contracts is their ability to automatically execute predefined actions when specific conditions are met. For instance, a smart contract could automatically transfer funds when a certain date is reached, or when multiple

1 https://docs.metamask.io/

parties provide their approval. This automation, not only reduces the potential for human error or manipulation but also significantly increases the speed and efficiency of transactions. In the context of healthcare, smart contracts could be used to manage patient consent, automate insurance claims, or securely share medical records while maintaining an immutable audit trail of all interactions [34].

Figure 4 showcases the relationships between these concepts (Smart Account and Smart Contracts).

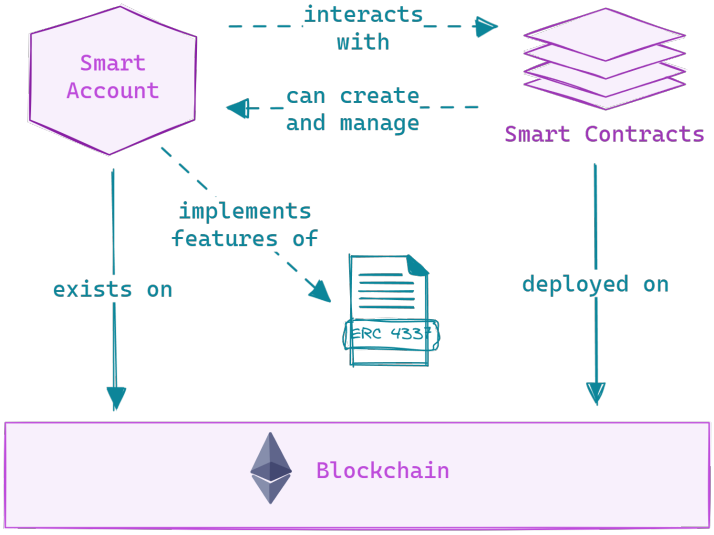


Figure 4: Blockchain, Smart Accounts and Smart Contracts

2.5 ZERO-KNOWLEDGE PROOFS

One of the most important ecosystems in blockchain today is based on the Ethereum blockchain and applications that use the Ethereum development stack. Ethereum is the first, and by far, the most used blockchain that has Turing-complete capabilities [35]. One of the advantages of Ethereum, over Bitcoin, is that it allows the execution of smart contracts. The biggest challenge for Ethereum growth is scalability, since currently limited scalability results in high prices of transactions [36]. One way of increasing scalability in the process reducing price is the use of Zero-Knowledge proofs (ZKP). The main purpose of the blockchain system is to execute transactions and smart contracts, for which users pay certain financial fees. Costs per transaction depend on the executed operations on the blockchain system, and the amount of used resources (primarily memory usage). Ethereum has a transaction execution limit per second between 15 to 30 Transactions Per Second [37]. This introduces another challenge: The blockchain can get clogged. Delays in transactions and code

execution, can be critical to high demanding operations, thus leading to severe failures. These operations are executed in the so called first level of the Ethereum blockchain or Level 1 (L1).

To address these challenges and enhance the scalability of the blockchain, a second layer, known as Level 2 (L2), has been introduced. L2 solutions are designed to operate atop the L1 layer, employing algorithms that aggregate multiple transactions off-chain before anchoring them back to the L1 layer in a single, bundled transaction. This process significantly reduces the strain on the L1 layer by decreasing the overall number of transactions that need to be processed directly on the blockchain. By abstracting and batching transactions, L2 solutions offer a scalable and efficient mechanism to increase the transaction throughput of the Ethereum network, ensuring faster processing times and lower costs, while also maintaining the security and decentralization benefits of the L1 layer. The integrity and security of transactions across both Level 1 (L1) and Level 2 (L2) layers, on the Ethereum blockchain, hinge on robust mechanisms for data protection and validation. This system involves a collaborative effort among distinct roles: the Coordinator, the Prover, and the Verifier. Initially, the Coordinator is responsible for collecting transactions executed on L2 and grouping them into a batch. This batch is then processed by the Prover, who generates cryptographic proof of the transactions' validity. Subsequently, the Verifier's role is to examine and affirm the correctness of this proof, ensuring that the transactions can be confidently updated on the L1 blockchain. The efficiency of this orchestrated process, directly influences the transaction execution speed of the multilayer blockchain system, highlighting the critical nature of each role in maintaining blockchain security and efficiency.

In the domain of Zero-Knowledge Proofs (ZKP), there are principally two distinct categories of algorithms: **Interactive** and **Non-interactive**.

- Interactive Zero-Knowledge Proofs: require a series of communication steps between the Prover and the Verifier throughout the validation process. This interaction is foundational to the protocol, as it ensures the integrity and validity of the proof. However, the requirement for ongoing communication can introduce complexities in certain applications.
- Non-Interactive Zero-Knowledge Proofs: present a more streamlined approach where the proof is constructed independently by the Prover, and can then be verified by any Verifier without further interaction. This trait renders non-interactive ZKP particularly advantageous for Blockchain applications.

Their inherent efficiency and autonomy align well with the decentralized and trust-minimized nature of the Blockchain.

This document, however, concentrates on the non-interactive variant, specifically on zkSNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge). Non-Interactive Zero-Knowledge Proofs (zkSNARK) enable the Prover to convince the Verifier of the truthfulness of a statement, without revealing the actual information behind the statement, nor requiring any direct interaction between them. This technology is instrumental in enhancing privacy and scalability within the Ethereum blockchain. It allows complex operations to be verified quickly and without disclosing sensitive details, which aligns with the broader objectives of maintaining account balance and transaction integrity across L1 and L2 layers.

2.6 ZERO-KNOWLEDGE SUCCINCT NON-INTERACTIVE ARGUMENT OF KNOWLEDGE

Also known as zkSNARK, a cryptographic protocol that allows someone, or something, to prove its possession of certain information without revealing the information itself. In order for this to work, zkSNARK implements three algorithms, each playing a pivotal role in the protocol's functionality [38], [39]. These components and algorithms can be seen on Figure 5 and can be described as such:

1. **Key Generator (G):** This algorithm takes a secret parameter, λ (lambda), along with a computational program, C, to produce two publicly available keys: a proving key (pk) and a verification key (vk). These keys are vital public parameters, crafted once per program C, and facilitate the subsequent proving and verification processes.
2. **Prover (P):** Given the proving key (pk), a public input (x), and a private input, known as the witness (w), the Prover algorithm generates a proof (prf Ppk, x, w). This proof substantiates that the prover possesses a witness (w) that adheres to the conditions set by program C, without disclosing any information about w itself.
3. **Verifier (V):** This algorithm evaluates the proof using the verification key (vk), the public input (x), and the proof (prf). If the proof aligns correctly with the program's conditions—indicating the prover indeed knows a valid witness (w)—the algorithm returns true, otherwise false.

These components and algorithms can be seen on Figure 5.

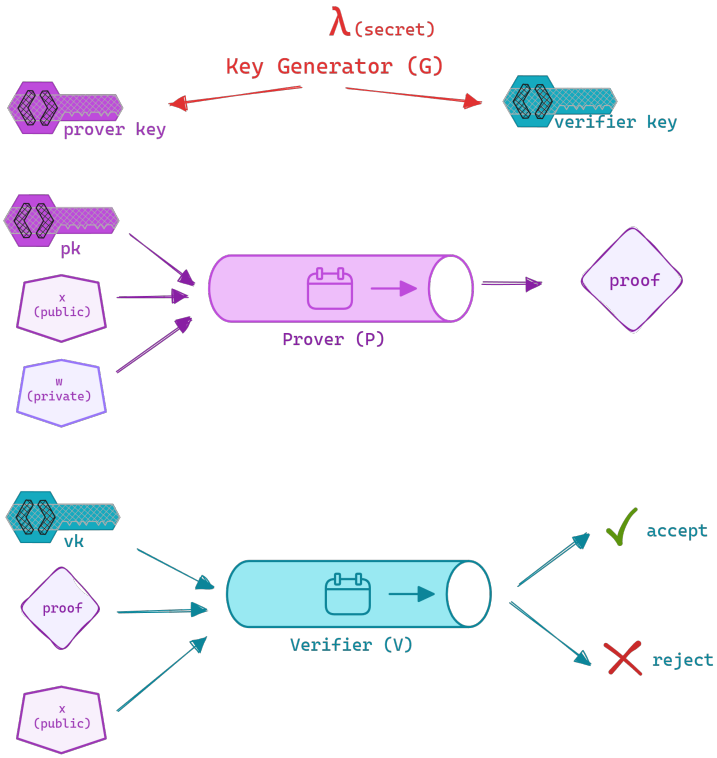


Figure 5: zkSNARK Algorithms Concepts

A critical aspect of zkSNARK is the secret parameter λ used during the key generation phase. The integrity of this parameter is paramount; exposure of λ could compromise the entire system. An entity in possession of λ could fabricate proofs `(fake_prf)` that passes verification without actually knowing the requisite secret witness (w). This potential vulnerability requires a highly secure and meticulously controlled key generation process [38].

2.7 ELETRONIC MEDICAL RECORDS

Eletronic Medical Records (EMR) are no more than the digital equivalent of any type of clinical data. This can be a representation of raw health values indicators, medical *Key Performance Indicators* (KPI), charts and any other metrics used in the clinical and healthcare industry. This is often used to build a background/history of the patient, with the aim of producing a more effective diagnosis, in a shorter period of time. Thus eliminating the waiting time between diagnosis and treatment. Ultimately, EMR are designed to integrate and streamline the patient care process, making information available instantly and securely to authorized users. These type of records are very wide, and can be spread across different, but similar concepts, namely

Personal Health Records (PHRs), Digital Medical Records (DMRs), Computerized Patient Record (CPR), Health Information Systems (HIS) and Electronic Health Records (EHRs). While often used interchangeably with EMRs, the last, EHRs typically encompass a broader scope of a patient's health history and are designed to be shared across different healthcare settings. EHRs include information from all the clinicians involved in a patient's care and authorized to contribute to the patient's health record, therefore, it is defined in the ISO/TR 20514 standard [40].

2.8 STANDARDS IN EMR'S

Like in every other digital transformation, there is the need of standards, with the healthcare sector not being an exclusion but in-fact one of the sectors in which ensuring interoperability, security, and efficiency is crucial to uphold a system with minimal to no errors. These standards are developed by national and international organizations, and are designed to ensure that EMR systems can exchange and make use of information cohesively.

Some of the key standards are the Health Level Seven International (HL7) [41] and Fast Healthcare Interoperability Resources (FHIR) [42]. Developed by HL7 organization, a set of international standards for the exchange, integration, sharing, and retrieval of electronic health information, FHIR is the standard that describes how the data is structured (formats), it's elements (resources) and how the data transaction occurs, namely, implementing a set of API's (Application Programmable Interfaces). Ultimately, FHIR aims to achieve the already well-know standards used in the communicationn between information systems, such as: high availability, easily discoverable, universally understandable syntax and the support across multiple platforms and applications, including: EHR's, mobile, big data, cloud computing and overall research and clinical trial ecosystems.

As opposed to HL7's FHIR technology and standards, there are other players in the scene, one of them being openEHR [43]. OpenEHR is an open-source, vendor-agnostic specification for EHR's and related health information technology (HIT) systems. It was designed to support the development of flexible, interoperable, and scalable EHR systems that can adapt to evolving clinical needs and technologies [44]. Fundamentally, openEHR establishes a collection of guidelines and standards for the structured, standardized representation and archiving of health information. As part of this, standardized clinical data elements—also known as archetypes—can be utilized to record and depict clinical concepts and actions are defined. These

archetypes can be combined to create conventional electronic health record templates that can be tailored to the unique requirements of various specializations and clinical situations [45]. Other healthcare standards are [46]:

- **SNOMED CT (Systematized Nomenclature of Medicine—Clinical Terms):** A comprehensive, multilingual clinical healthcare terminology that provides precise coding of data related to healthcare. It enables consistent data capture, sharing, and aggregation across health systems [47].
- **LOINC (Logical Observation Identifiers Names and Codes):** For lab tests and clinical observations, LOINC provides a universal code system for identifying health measurements, observations, and documents [48].
- **HIPAA (Health Insurance Portability and Accountability Act) in the United States:** While not a technical standard, HIPAA sets the standard for protecting sensitive patient data. Any company that deals with protected health information (PHI) must ensure that all the required physical, network, and process security measures are in place and followed [49].

2.9 RELATED WORK

The European Blockchain Services Infrastructure (EBSI) [50] is a pioneering initiative by the European Commission and the European Blockchain Partnership (EBP), comprising 29 countries (EU Member States, Norway, and Liechtenstein) committed to facilitating cross-border services between governments, businesses, and citizens using blockchain technology. EBSI aims to increase the efficiency, security, and transparency of online services across the European Union (EU) and associated countries, leveraging the inherent benefits of blockchain technology to foster trust, reduce fraud, and cut costs. Under the hood, EBSI operates upon an implementation of the Hyperledger Besu [51], making it a permissioned-based blockchain. The onboarding demonstrates that there is a central authority that issues credentials to issuers, eg. Ministries, to provide the necessary authorization and accreditation of the verifiers and it's users. This of course, leads to a stricter flow of the operations, as not only it ends up having a central authority, which is the European Commission itself, but each it's members have their own control over their nodes, therefore going against to the decentralization of powers [52]. Currently, EBSI is more focused on use cases related to the accreditation of diplomas, across its' members, as there are some pilot projects ongoing. Along side, there is a focus towards the digitization of government services with the use of the blockchain, whereas, one key factor, and

that EBSI mentions a lot is the decentralized ID, which acts as a public identifier of an entity represented in the chain [22].

MedRec [53] holds the first spot when pioneering a solution for managing EMR's based on blockchain, enhancing patient control over their medical history through a comprehensive and credible log. It enables granular, patient-initiated data exchange across medical jurisdictions with specific authorization controls for data confidentiality, supported by smart contracts for time-bound access rights. Integrating with existing healthcare data infrastructures, MedRec promotes adoption while aligning with HIPAA [49] regulations for interoperability, supporting standards such as FHIR [42] and HL7 [41]. Its decentralized model ensures data is stored across multiple nodes, reducing risks of centralized data breaches, though it does not address individual database security or Digital Rights Management (DRM) issues [54]. Since MedRec [53] relies on an early blockchain implementation, and relies solely on it, it faces two main issues: First, a blockchain only stores immutable records, which is a limitation in the health scenario because the actual locations of data are likely to be changed. When a URL is modified, a new block containing the new URL has to be generated, rather than modifying the old block (due to content immutability of blockchain). Second, their data sharing schemes do not reclaim the space after a sharing. Thus, a data sharing session also takes a new block [16]. Both of these limitations result in high storage overhead.

Medchain [55] proposed a healthcare data exchange system that relies onto two separate decentralized networks: an immutable ledger that withholds the metadata (fingerprint) of the patients information, session, and operations; a second peer-to-peer (P2P) storage network, that as the name suggests serves as the actual storage of the data, where descriptions of the data and the session are persisted. The difference among both is that the data is mutable, thus can be updated. This raises the authorization, authenticity and integrity of the system, so in order to achieve security and preservation of privacy, Medchain [55] uses a session-based data exchange.

FHIRChain [56] presents an approach to enhancing clinical decision support and achieving interoperability in healthcare IT systems through the integration of blockchain technology and FHIR [42] data standards. This prototype addresses critical Office of the National Coordinator for Health Information Technology's (ONC) [57] interoperability roadmap requirements, including user identifiability and authentication, secure data exchange, permissioned data access, consistent data formats, and system modularity. FHIRChain [56] yielded several important insights. Firstly, the system provides a trustless, decentralized storage solution

for meta information and audit logs, effectively overcoming issues of proprietary vendor lock-in by facilitating secure, scalable clinical data sharing without the need for established trust among healthcare participants. Secondly, FHIRChain [56] enhances data exchange efficiency while maintaining data ownership through the creation of lightweight reference pointers, rather than transferring actual data, which is particularly advantageous for telemedicine and clinics in network-limited rural areas. Lastly, the application of public key cryptography for digital health identity management emerged as a viable method for secure data sharing, simplifying authentication and ensuring data integrity, though it poses challenges in managing private keys for general patient use.

3

PROPOSED ARCHITECTURE

As described before, the objective of this document is to research and propose a solution for the following scenarios: identity (see Appendix E) and access management (see Appendix F). To achieve confidentiality, patient identity and the existence of patient-provider relationships cannot be publicly disclosed, nor allowing third party entities to perform queries in the blockchain. Being able to formally bound two entities together without revealing the existence of it to others, requires some technological challenges, thus a more conscious decision for this solution is required. Here's where the concept of proving something that only the parties involved know, without revealing any information seems to be very relevant. This can be achieve by implement a solution based in Zero-Knowledge Proofs (see Section 2.5).

By integrating Zero-knowledge proofs (ZKP) within the blockchain, patients can engage with healthcare providers in a secure, private environment, where only mutual consent reveals the connection between them. In this model, a patient's health records and personal identity details are safeguarded from public exposure, while still being accessible to authorized healthcare providers, through a secure and private channel. When a patient initiates a relationship with an healthcare provider, the so called "Data Sharing Agreement", it grants access through a ZKP mechanism that verifies the patient's consent, and the provider's legitimacy, without disclosing any identifiable data on the blockchain. Only involved parties — the patient and the healthcare provider — are aware of the established relationship, and can prove its existence to each other, ensuring that there is no digital footprint that could be exploited to reveal their connection.

Other major advantages of this patient-centric approach are the allowance of restricting the information shared with the healthcare provider. The patient is now able to share only the information that he suits best for each interaction between the him and the healthcare provider. This gives patient more control over their data and is a step forward for the patient to be the principal owner of their data.

To portray the proposed platform's structure, the C4 model [58] was chosen due to its ability to offer a coherent and layered visual representation. It allows for the illustration of fundamental components, their interconnections, and their interactions

within the proposed system in an orderly fashion. This clarity benefits all project participants, from developers to project managers, by aiding their understanding of the platform's organization and operational dynamics. It facilitates effective dialogue by establishing a common language for architectural and design discussions, minimizing the risk of confusion and misinterpretation. Consequently, everyone engaged in the project shares a unified understanding. The C4 model is set to be an instrumental guide for the development process, as well as for the documentation and articulation of the platform's architecture.

3.1 OVERVIEW OF THE ARCHITECTURE

First and foremost, having a higher abstract view of the architecture allows the reader to better understand the overall system and its structure, so the decision of creating a Level 1 C4 model [58] seemed more than adequate, as shown in Figure 6.

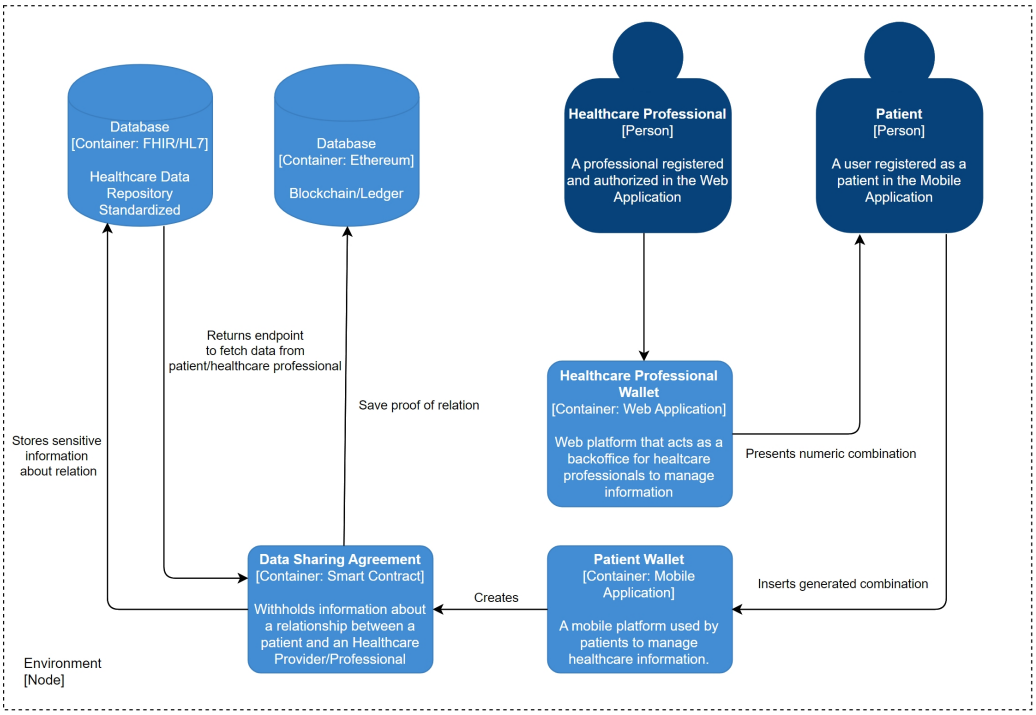


Figure 6: Proposed system architecture

The architecture is distributed as the following:

- **Node**:
  - A database container for storing healthcare-related data.
  - Use of FHIR/HL7 standards for healthcare data interchange.

  - Provides an API or service for data retrieval.

- **Blockchain/Ledger**:
  - A database container for blockchain or ledger data.
  - Stores immutable proofs of relations between patients and healthcare professionals.
- **Healthcare Professional Wallet (Web App)**:
  - A web application used by healthcare professionals.
  - Serves as a back-office tool for managing patient information.
  - Accessible to authorized professionals registered within the system.
- **Patient Wallet (Mobile App)**:
  - A mobile platform used by patients to manage healthcare information.
  - Interacts with the system by inserting a generated numeric combination.
- **Data Sharing Agreement (Smart Contract)**:
  - A smart contract container on the blockchain.
  - Manages data sharing agreements, while ensuring privacy and security.
  - Records proofs of relationships on the blockchain/ledger.

The arrows between the containers in the C4 Level 1 diagram represent data flow or interactions:

- **Node**:
  - Returns an endpoint to fetch data, indicating an API or service that allows data retrieval from a patient/healthcare professional relationship.
- **Blockchain/Ledger**:
  - Interacts with the Data Sharing Agreement to save proof of the relationship, indicating that proofs are securely stored on the blockchain/ledger.
- **Patient Wallet (Web App)**:
  - Inserts a generated combination into the system, which could be related to two-factor authentication or a unique identifier for secure access.
- **Healthcare Professional Wallet (Mobile App)**:
  - Presents a numeric combination, likely related to authentication or data access mechanisms.

3.2 A COMPREHENSIVE REVIEW ON THE ARCHITECTURE

Having established the foundational overview of the system's architecture in the preceding section, we now transition to an in-depth exploration of the Level 2 architecture as outlined by the C4 Model. This section aims to dissect some of the containers within the system, providing a granular view of each application, data store, and service that collectively form the backbone of this platform's functionality.

The C4 Model's Level 2 architecture examines the interrelationships and interactions between the containers that were introduced at Level 1. An in-depth analysis of the roles that the containers perform, the technologies they use, and the conventions they follow can be achieved by digging into the specifics of each one.

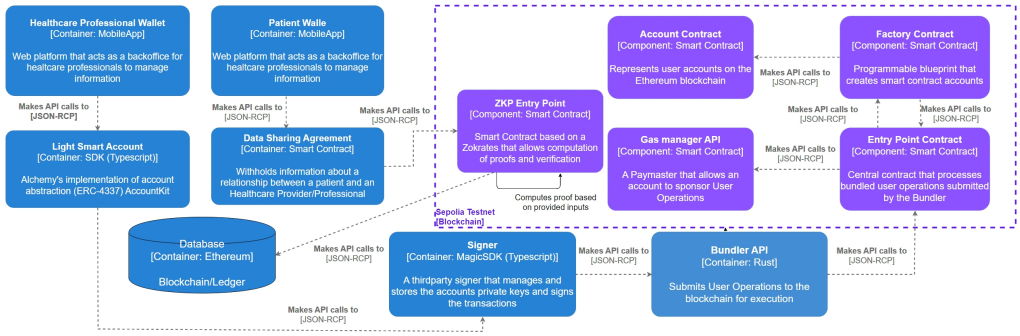


Figure 7: Proposed system architecture extended to C4 Level 2

3.2.1 *Containers*

- **Healthcare Professional Wallet [Mobile App]**: Serves healthcare professionals as a management interface for patient data. It interfaces with smart contracts to manage consent for data sharing.
- **Patient Wallet [Mobile App]**: Provides patients with an interface to manage their personal healthcare information and control sharing preferences.
- **Light Smart Account [SDK/TypeScript]**: Implements ERC-4337 for account abstraction, handling authentication and authorization of patient accounts.
- **Bundler API**: The Bundler API is a Rust-based component that takes signed User Operations and submits them to the blockchain. This process involves

compiling transactions into a single batch for efficient processing, which is crucial for optimizing network and miner fees.

- **Signer**: Using MagicSDK, the Signer is responsible for securely managing cryptographic keys and signing blockchain transactions. It ensures that user credentials are managed safely, adhering to non-custodial standards, and supports the system's overall security posture.
- **Data Sharing Agreement [Smart Contract]**: Manages agreements between patients and healthcare providers, safeguarding the conditions under which patient data is shared.
- **Database [Blockchain/Ledger]**: An Ethereum blockchain ledger that persists transactional data and smart contract states.

3.2.2 *Components*

- **Factory Contract [Smart Contract]**: Deploys new Light Smart Account contracts with pre-defined logic and configuration.
- **Account Contract [Smart Contract]**: Represents the patient's account on the blockchain, detailing its logic and interaction rules.
- **Entry Point Contract [Smart Contract]**: Acts as the router for the user operations, interfacing with the Bundler API to validate and execute transactions on the blockchain.
- **ZKP Entry Point [Smart Contract]**: Uses zero-knowledge proofs to validate transactions in a privacy-preserving manner, without revealing the contents to the blockchain.
- **Gas Manager API [Smart Contract]**: Manages the gas fee sponsorship for user operations, enabling a better user experience by abstracting the complexity of gas fees.

3.2.3 *Interaction Flow*

The system's structure is illustrated by the interactions between its major components, as depicted in Figure 8

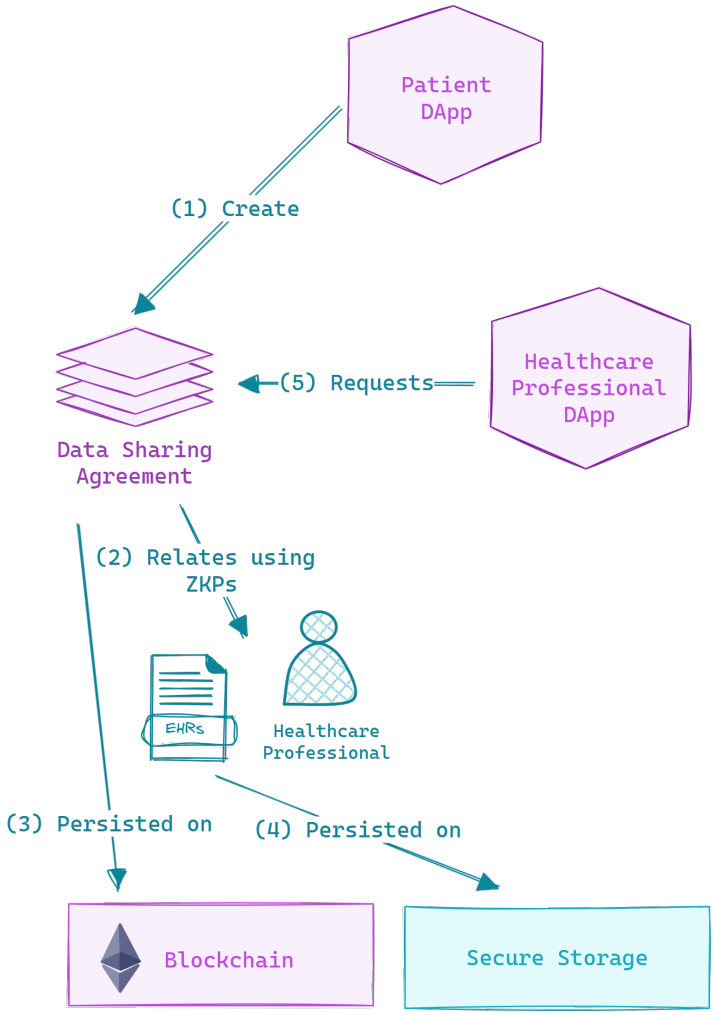


Figure 8: Interaction Flow

The patient Dapp can create a Data Sharing Agreement (Step 1) that relates EHR and healthcare professionals (Step 2). This relationship is the contribution of this document and is performed using ZKP in order to ensure that the relationship itself is not exploitable.

The Data Sharing Agreement is publicly persisted on a public blockchain (Step 3), since it does not contain any identifiable data to anyone outside of the actors (that took part of the creation of the ZKP). The actual EHR are securely stored off-chain using some form of cryptography (Step 4).

When a healthcare professional needs access to the EHR, they use the Data Sharing Agreement (Step 5), which ensures the ZKP is valid, and uses that proof as an authorization mechanism to access the Secure Storage where the actual data resides.

3.3 TECHNOLOGICAL DECISIONS

The implementation of the architecture is underpinned by a selection of cutting-edge technologies, each chosen for its robustness, flexibility, and suitability to the platform's requirements. Below we delineate the key technologies and their roles within this system.

3.3.1 *Mimicking the blockchain envrionment*

In order to implement such solutions there is the need to setup a free of charge working environment that replicates the Ethereum blockchain itself [20]. Since the start of Smart Contract (see Section 2.4) developing there had been some implementations and developer tools designed to withstand the need of building decentralized applications.

Tools like Truffle [59], Hardhat [60] or even the Remix [61], are some of the choices available for developing backend applications in the Ethereum Virtual Machine (EVM) [62]. Truffle is becoming not so popular nowadays, as Hardhat rises. So the choice was simple, use Hardhat for setting up a local blockchain environment.

Hardhat is an Ethereum development environment that facilitates building, testing, deploying, and debugging of smart contracts. It allowed out-of-the box configuration for local-network blockchain, where no real fees are applied in the transactions, thus allowing a faster development process and testing. Another great feature is the ability to deploy the developed code in a so called *Test Net* [63]. These are actual blockchain networks that replicate Ethereum's *Main Net*, allowing developers to test their smart contract code in a «real» environment, further allowing the simulation of the costs, fluidity and other Key Performance Indicators (KPI).

The chosen *Test Net* was Sepolia [64]. The selection of a test network is not a hundred percent up to the developers as its something that relies on the infrastructure and what network is being deployed as a testing environment by the blockchain engineers. In this project's timeline, Sepolia was the most suitable, being the one mentioned and sustained by Alchemy's guidelines [65]. Something to take note is that: these test networks don't offer limited "credits" or *gas* to pay for the fees. For that, a *faucet* is used to provide some currency token. Alchemy's *faucet* only provided 0.5ETH per day, so most of the development is done locally with Hardhat local network, and then tested in the Sepolia network.

3.3.2 *Enhanced blockchain interaction*

The *AlchemySDK* [66] serves as a bridge between the frontend application and the Ethereum network, powered by Alchemy's infrastructure. It simplifies interactions with blockchain, providing reliable access to on-chain data and scalable infrastructure for submitting transactions. It is particularly vital for managing the complexities of account abstraction provided by Alchemy's Account Kit.

Using the *account abstraction* features from Alchemy enables, patients to create a Light Smart Account, which facilitates a more flexible and user-centric experience. This technology abstracts away the traditional complexities associated with Externally Owned Accounts (EOA) [8], such as managing cryptographic keys, sigining transactions, paying gas fees, and so on.

3.3.3 *User interface*

To implement the proof of concept (POC) addressing the use of Account Abstraction to provide a more enhanced user experience (UX) for the patients, a web application was implemented to replicate the Patient Wallet. Since Alchemy's documentation was rich in TypeScript[1] and React[2] examples, the choice was clear. NextJS[3], a React framework that provides all the tools for building a client application that can be rendered in numerous ways.

This section won't prolong itself in describing what this frontend framework offers, but instead addresses all the necessary features to support the concept addressed in this thesis. A worthy mentioned goes by saying that the web application is intended to prove that the use of account abstraction can be used to implement features, already present in Web2.0, such as fluid authentication workflow that does not required the self-management of cryptographic keys. This document will go into more details in Chapter 4.

1 https://www.typescriptlang.org/
2 https://react.dev/learn
3 https://nextjs.org/docs

3.3.4 *Third-party signer*

As the Signer's role was already described, there is the need to provide some way to remove the hassle of managing cryptographic keys and signing each transaction. After carefully reviewing available third party offers, the decision was to use Magic [67], for simplifying the authentication process for patients.

Magic provides multiple authentication mechanisms, such as OAuth2.0 [68] where it implements social logins as well [69], so that the user can register himself with Google, Facebook or any other social accounts they already have. It also ships some components out-of-the-box compatible with the chosen frontend framework. Another crucial benefit is account access recovery.

To enable this using Magic, the wallet must register a second authentication method using a phone number. In addiction the wallets must be configured as "Dedicated Wallets" which is only available in the *Enterprise plan*. Furthermore, Magic's Wallet implementation follows the Non-custodial model.

These types of wallet providers keep private keys in a way that prevents them from being accessed without the user's consent. For instance, in order to decrypt the private key that the provider has stored, the user needs to supply a password or passkey that only they know. Users gain from increased security while always maintaining control over their private keys. This is comparable to a safety deposit box vault in which only the user has access to the individual safety deposit boxes (such as wallets), while the provider guards the bank vault.

3.3.5 *Zero-knowledge proofs implementation*

Implementing a Zero-Knowledge Proof (ZKP) solution, compatible with the Ethereum Virtual Machine (EVM) that can be deployed as a smart contract and easily integrated with javascript can be challenging. Prior to research, the choice of ZoKrates [70], a toolbox designed to facilitate the development of ZKP applications on the Ethereum blockchain was made. It provides a high-level language to write private computations, a compiler to turn them into a set of arithmetic constraints, and a trusted setup to generate verifiable proofs that are compact and efficiently verifiable within the EVM environment.

This integration allows for the construction of decentralized applications that require verification of certain conditions without revealing any sensitive information, thereby enabling enhanced privacy and security features in smart contracts.

4

DEVELOPMENT

Solving the challenges of identity management and access control in healthcare systems required extensive research into complex subjects involving cryptography, mathematics, and computational science. The identity management issue revolves around ensuring that patient identities and their relationships with healthcare providers are securely managed without revealing sensitive data. On the other hand, access management focuses on giving patients full control over who can access their health information, ensuring that unauthorized parties, including data mining algorithms, cannot link or exploit the data. Addressing both of these problems demanded innovative approaches and advanced technological solutions.

After carefully reviewing all the components needed (see Appendix E and F), some ideas were brainstormed and presented as possible solutions to mitigate the challenges related to identity and resource management. Next sections will go in depth on all the development phase, the challenges faced, how they were overcome and the ending result of a working Proof of Concept (PoC).

4.1 SOLVING THE ACCOUNT MANAGEMENT PROBLEM

The identity (Account) management problem stems from the need to securely establish and manage these relationships without exposing any sensitive information. Patients must be able to authenticate themselves and interact with healthcare providers while maintaining control over their personal data and ensuring it cannot be exploited by unauthorized third parties. From the perspective of technical research done in this document, there are three viable solutions for addressing the identity management problem, which are represented in E and can be described as such:

1. **Master Account**: Whereas the master accounts holds it's own mapping for the mapping of the Patient (P) identifier and the Healthcare (HC) identifier *(master -> relation(P,HC))*;
2. **Multiple relations** : Each Patient Wallet holds and manages each of the relations between the user and the Healthcare Provider;

3. **Account Abstraction and *Zero-Knowledge Proofs***: Proofing that there is a relationship between two identifiers (User and Healthcare Provider) without exposing any private data from both parties.

From the three viable proposed solutions to solve the mapping problem between accounts, the last one seemed to be the most interesting, both technologically and scientifically. The value of a fully working solution using *Zero-knowledge proofs* is groundbreaking in this subject, as it can improve the security, control and privacy over the data shared amongst parties. Before proceeding to explaining this last method that combines both Account Abstraction (ERC-4337) [71] and Zero-Knowledge Proofs (ZKP) [72], the other ones must be address on to why they don't seem the most fit candidates. The first one, Master Account, relies on having a central authority that withholds every relationship that a patient has with corresponding Healthcare Providers. This sentence provides the answer on its own. This approach seems against the core concept of using the blockchain, which is to achieve decentralization. Needless to say, having such central authority requires supervision as to what ends up in the master account, exposing the identity of all the entities involved in the mapped relationships.

On the other hand, having each Patient Wallet managing their own relationships (Multiple Relationship model, see 2) sounds like a viable solution, only when not taking into consideration the fact that every relation must be recorded somewhere, and this "obscure" place should be the ledger. This allows auditability of data and integrity as well. Although this solution benefits from having data placed in somewhere, that can be easily audited and verified, it raises the biggest concern: the information is public to all participants of the ledger. Since this scenario deals with a public blockchain like Ethereum, privacy related issues invalidate the use of this approach. One key requirement to solving these problems is to also give anonymity and avoid untrusted authors to use the publicly available information, mainly applying data mining algorithms and infer that Patient X has a relationship with Healthcare Provider Y.

Since, the proposed solution to mitigate the identity management problem should rely on minimum interaction and user data, Non-Interactive based Zero-Knowledge Proofs (zkSNARK) seem to be the most adequate candidate for Patient's Wallet use case (see Section 2.6).

There are several implementations of non-interactive algorithm's, mainly: zk-SNARK, zkSTARK [72], PLONK [73] and Bulletproofs [74] which are explained in 2.6.

After carefully discussing and reviewing the requirements of the system, the zkSNARK was the fittest option as it offers low gas fees, decent and primarily due to the availability of the ZoKrates [70] toolkit, which integrates seamlessly with Solidity, the programming language for Ethereum Smart Contracts. ZoKrates offers a comprehensive suite for creating and verifying zkSNARK, making it a preferred choice for blockchain-based applications, where privacy and efficiency are key concerns.

4.2 BLOCKCHAIN DEVELOPMENT ENVIRONMENT

Mimicking Ethereum's blockchain environment is one of the most important step towards the success of this proposed solution. Some tools like Truffle/Ganache [59], [75], Foundry [76] or Ape Framework [77] were considered, and all provide a similar and easy configuration to quickly setup a local test network to interact with smart contracts and simulate the ledger. The key factors for choosing the right Ethereum development platform were: the support from both the community and the maintainers of the framework, the interoperability with others, such as testing tools like Mocha[1] and Chai[2] and a strongly typed programming language like Typescript [78]; compatible with the latest tech stacks and Web3 plugins and API's and the *out-of-the-box* setup. The winner was clear, Hardhat [60]. This Ethereum development framework, not only was emerging at the time this Proof of Concept (PoC) was being implemented, which means recent codebase and compatibility with the more recent changes in Web3 available Toolkits and API's, but also the exponential growth of it with multiple developers working and developing on it. Another major advantage over the others was that Hardhat is supported by the Ethereum Foundation.

4.3 CREATING THE WALLET

Before proceeding to implement the more complex solution that uses Account Abstraction to configure Smart Accounts for the users, a simple Externally Owned Account (EOA - see 2.4) [8] has to be created with the sole purpose of development only. This account will be linked to the Alchemy platform (and others), and is going to be the account that deploys all the artifacts necessary onto the testing blockchain

1 https://mochajs.org/
2 https://www.chaijs.com/

network, in our case, Sepolia [64]. To quickly setup a Web3 Wallet, Metamask [79] offers a plugin or extension that can be installed in the browser and provides a convenient UI/UX to setup a Metamask Wallet.

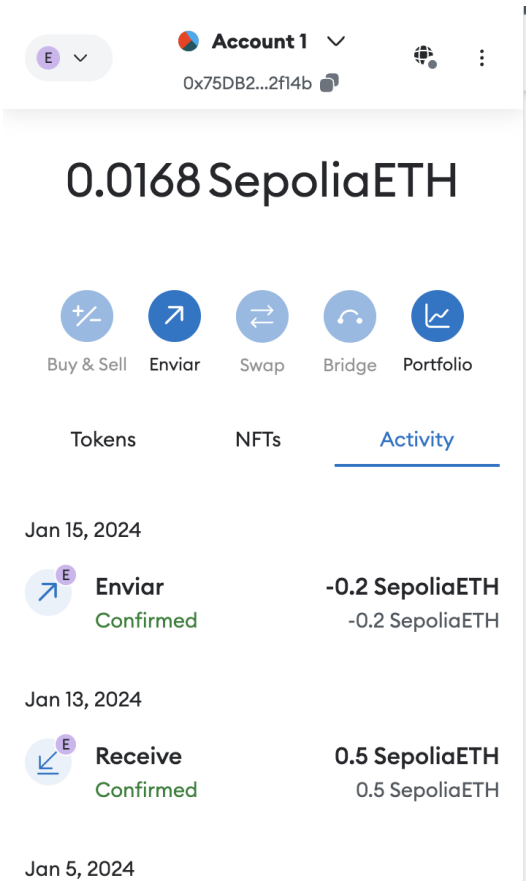


Figure 9: Metamask Wallet Extension

After onboarding in Metamask, the Wallet will have a similiar interface Figure 9 shows. In the top corner, at the center, there is the alias for the account, namely `Account 1`, which is automatically assigned when the user does not specify an account name (this is just for organization means, as the user may or not have multiple Metamask Wallets). Further down, there is one important token or address: `0x75DB2D6e602549AE60C53f41A24a4cBc49e2f14b`. This is the public address or `identifier` (`ID`) that is going to represent this EOA Account in the public blockchain, in the case study of the Sepolia test network.

There are multiple tools available online that allow the verification of the accounts created, the one used while developing this project is Etherscan [80]. It provides detailed information on any address registered in the Ethereum blockchain. An important consideration to take note of, is, that every test network has its own Etherscan domain.

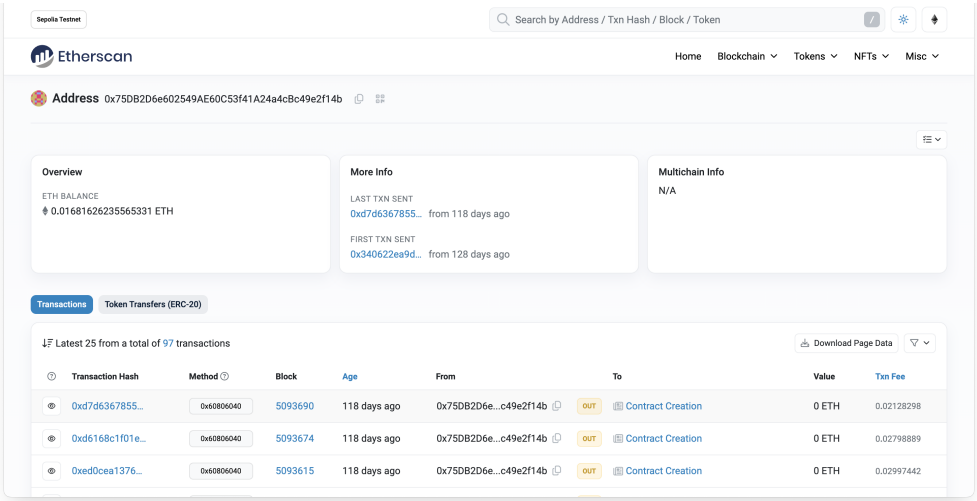


Figure 10: Metamask Wallet Address on Etherscan

This can be evident from Figure 10 as the current address starts with “sepolia” then the remaining domain of Etherscan: “sepolia.etherescan.io”.

The dashboard shown in Figure 10 also enables searching for the public address of the newly created EOA, providing a more detailed view of the wallet, including the transactions performed.

Every actor, with legitimate or malicious intentions, has access to every account registered and their balance.

Balance is the amount of ETH, Ethereum tokens, that an account detains. These tokens are used to pay for gas fees (see Section2.3) and to be used as actual currency. Outside the Sepolia Test network, this balance is worthless, but it represents the current value of the “main” currency and actual value of the Ethereum “coin”. Like all the other physical currencies, USD, EUR, etc; ETH can be exchanged. The current value for the amount/balance of ETH tokens in this Account is marked at 50.05USD. This value is volatile and therefore fluctuates overtime, so this is just an example to further understand the value and conversion rate of the ETH token.

Another critical aspect of EOA is the lack of private key management. This is one of the problems that Patient Wallet aims to solve. As mentioned before, in the traditional Web2.0 there are several methods to provide users a frictionless onboarding into applications, whether through social logins that use social media accounts, or other signin methods as passwordless or phone number. Currently, most of the Web3.0 (decentralized applications - dapps) use EOA’s which are complex and have multiple security flaws, starting by where to store the account’s private key

(see Section 2.2). It wouldn't be unprecedented that users lost their account private keys and no longer can access their account as EOA don't offer account recovery protocols by design. Numerous reports of blockchain users, mainly in the Bitcoin industry, have allegedly lost their private keys, either stolen or just negligence [81]. The user private key is critical because without it, the owner cannot claim ownership over itself. Not to mention, if a malefactor has control over a stolen private key and exploits it.

To avoid potential issues, particularly being connected to the wrong blockchain network, it is essential to verify that the Metamask extension is properly configured to use the Sepolia Network. Failing to do so may result in transactions being executed without appearing in the current development wallet, leading to difficulties in debugging later in the development process.

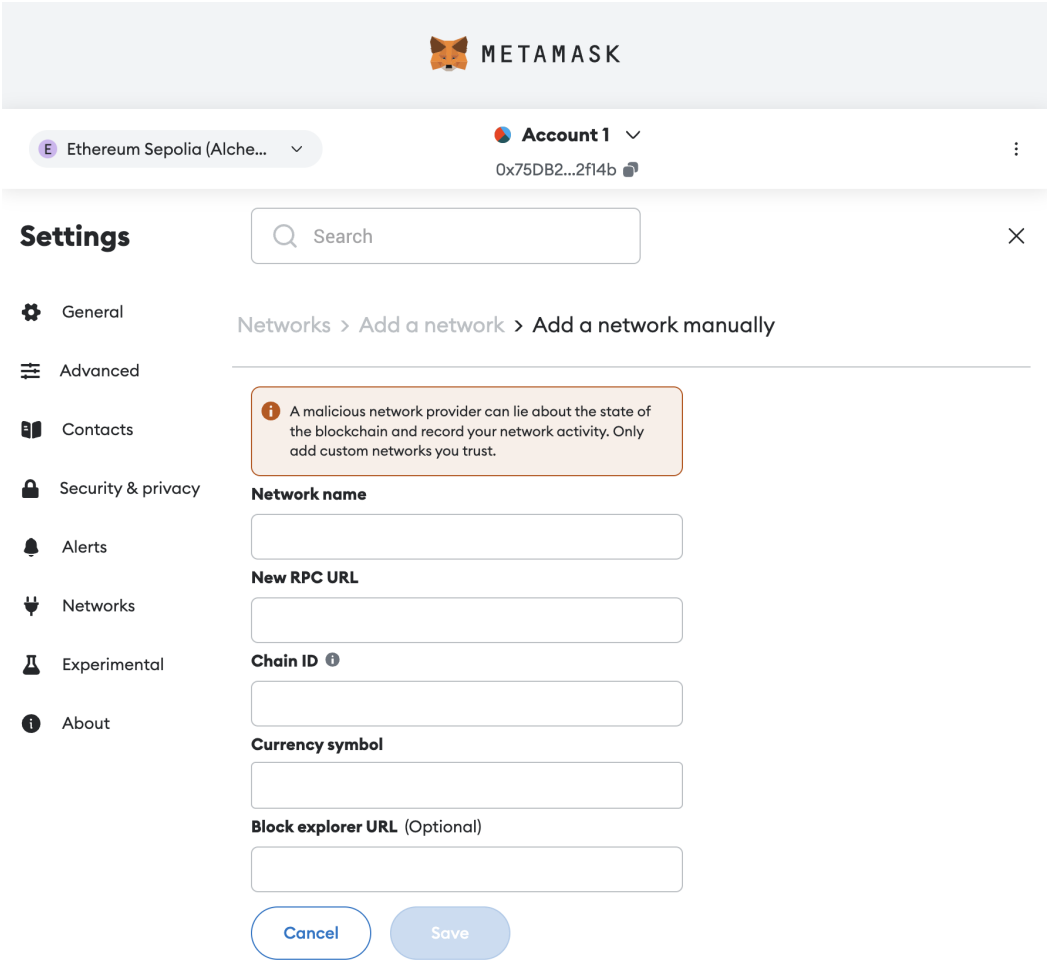


Figure 11: Metamask Wallet Network Configuration

The form fields, seen in Figure 11, except the last one, are mandatory and can be found in Alchemy's documentation[3]. The `Chain ID` identifies the Sepolia Test Network deployed in the Ethereum ecosystem. The `RPC URL` exposes an entry point that facilitates communication between the development environment and the Sepolia Test Network. Alchemy provides their own `RPC URL` and can be found in Alchemy's documentation[4]. Using Alchemy's RPC is preferred as it offers better transactions details and development features instead of using the official Sepolia `RPC URLs`. The currency symbol is the same as the parent/main network - ETH.

Finally, the Wallet must be credited with Sepolia testing tokens (ETH). This is the currency used to pay the gas fees for deploying smart contracts and making transactions on the test network. Although one might say this is redundant or why bother creating and fund an EOA when Hardhat provides a testing environment with endless funded accounts, this is relevant because the proposed solution must assert its correct functioning across all Ethereum based networks. Therefore, it is necessary to examine the behavior in a "real scenario". Since all the transactions are placed inside the Sepolia Test Net, short for network, the Wallet must be credited with some tokens to reach a positive balance. Alchemy's approach, and they are not the only ones, is to use a Faucet[5]. Also known as *crypto faucet*, it consists of a website or application that rewards the user with cryptocurrency for completing one or more tasks. In case of Alchemy's Sepolia Faucet, represented in Figure 12 the task is as simple as the user connecting his Alchemy account, corroborate the validity of a recaptcha - questionaire or task to prove that the user is not a computer pretending to be an actual human[6] and clicking on a "Send Me ETH" button.

3 https://docs.alchemy.com/docs/choosing-a-web3-network#sepolia-testnet
4 https://docs.alchemy.com/docs/choosing-a-web3-network#networks--their-details
5 https://www.ethereum-ecosystem.com/apps/alchemy-sepolia-faucet
6 https://developers.google.com/recaptcha

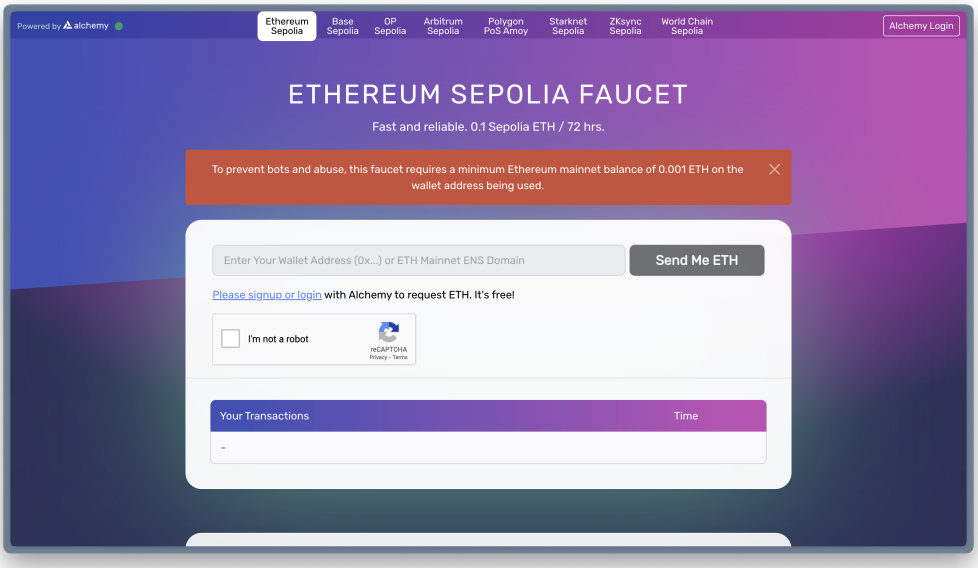


Figure 12: Alchemy Sepolia Faucet

In later stages of the development cycle of this project, a new prerequisite was filed by Alchemy which stated that every developer must detain at least 0.001 worth of "real" ETH in his Wallet to request more funding. This is groundbreaker to some, because this means that the developer must obtained real ETH tokens (main net) to continue using all the features that a Test Net provides. Furthermore, the amount of ETH supplied by Alchemy's faucet is limited to 0.5 ETH per day. As Alchemy states in their website, this is to prevent "bots" and abuse from unwanted users to exploit this feature.

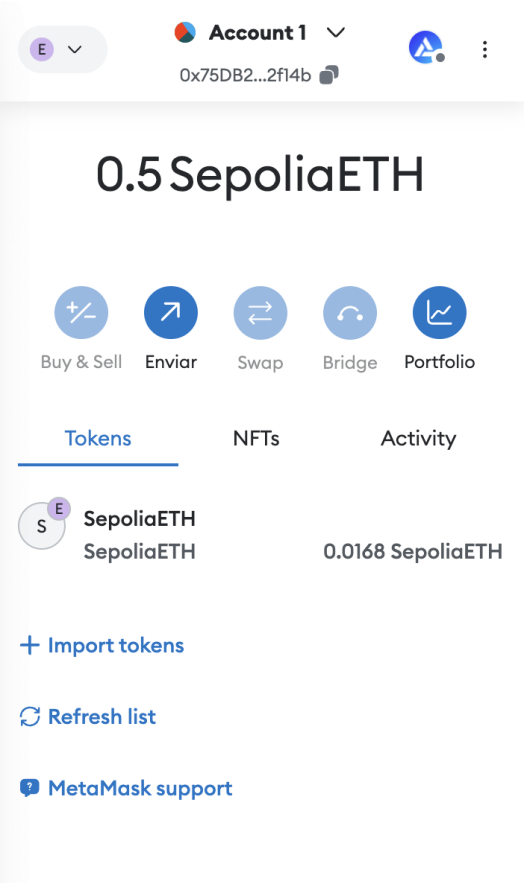


Figure 13: Wallet - Metamask

Finally, the user may now view the deposited funds in the Metamask Wallet like shown Figure 9.

4.4 SETTING UP A LOCAL DEVELOPMENT ENVIRONMENT

A crucial segment of this project is to achieve the closest conditions to what an in-production Ethereum blockchain represents. For this purpose, the tool chosen was Hardhat[7], as it provides all the necessary features required in this scenario.

Before embarking in the journey of configuring this tool, a clear definition of the workspace must be detailed. Therefore, the project's structure can be describe in Figure 14

7 https://hardhat.org/docs

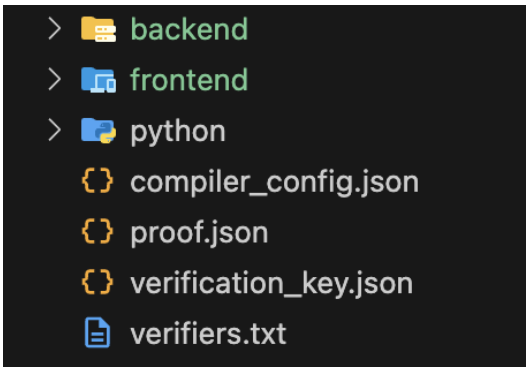


Figure 14: Project Structure

The main directory (Patient Wallet) consists of a `frontend/` folder which contains the Web Application that represents the Patient Wallet - The patient's platform to register themselves in the Patient Wallet, using Account Abstraction; a `backend/`‘ folder with all the logic behind the asset management of the patients resources as well as the contract used to verify the proof of relationships between patients and healthcare providers; a `python/` folder with earlier testing scripts of cryptographic key derivation methods. This last one will be further detailed throughout this document.

Expanding the `backend/` directory yields more information on how the backbones of this service is structured. This can be represented by Figure 15.

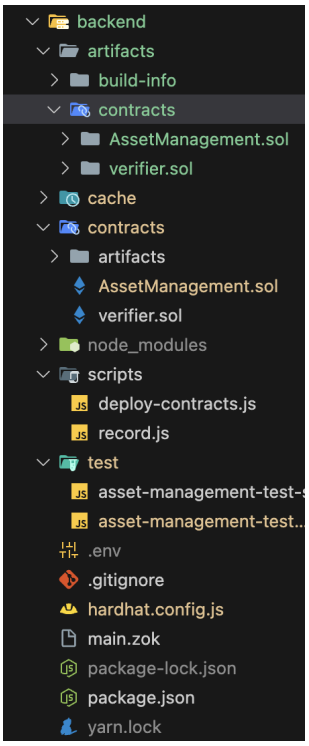


Figure 15: Backend Folder Structure

Seen in Figure 15, the `contracts/` folder is where all of the Smart Contracts are allocated, and can be easily identified by their extension `.sol`, which corresponds to the Solidity programming language, mentioned in Section 2.4. Following is the `artifacts/` folder, which is crucial to understand how contracts work and interact. This directory is automatically generated by Hardhat upon the compilation of the smart contracts (`SOL` files). Each compiled contract within the folder is represented by a `JSON` file.

All `JSON` files contain the Application Binary Interface (ABI) and the bytecode of smart contracts. The ABI outlines methods and structures within the smart contract. It specifies the callable functions, their parameters, and return types, alongside event specifications and other accessible components of the contract. This interface is indispensable for any external application, or in this use case, the frontend Web Application, that needs to interact with the smart contract, as it dictates how calls to the contract should be formatted and executed. The bytecode contained in these files represents the compiled and deployable version of the smart contract. This is the actual code that the Ethereum Virtual Machine (EVM) interprets and executes on the blockchain. The bytecode is what gets deployed to the Ethereum network, and thus forms the operational core of the smart contract.

4.5 MANAGING RESOURCES

The `AssetManagement` is the core for implementing the business logic of the asset/resource management system. As Figure 16 shows, the main elements of this contract include a *Patient* data structure, mappings for patient data management, access control modifiers to grant or revoke unwanted users from accessing the contract's code, and functions that execute upon the state of the contract.

```
pragma solidity ^0.8.9;

contract AssetManagement {
    // Variables
    struct Patient { address patientAddress; string[] medicalAssetIds; mapping(string => mapping(address
=> bool)) accessGrantedPerAssetId; }
    mapping(address => Patient) private patients;
    address[] private patientAddresses;

    // Modifiers
    modifier onlyPatient() {}
    modifier onlyAuthorizedForAssetId(address _patient, string memory _assetId) {}

    // Events
    event PatientRegistered(address indexed patientAddress, address indexed patient);

    // Functions
    function registerPatient(address _patient) public {}
    function addMedicalAssetId(string memory _id, address _patient) public {}
    function getAllAssetIds() public view returns (string[][] memory) {}
    function grantAccessToAssetId(string memory _assetId, address _entity) public onlyPatient {}
    function viewAllAssetIds() public view onlyPatient returns (string[] memory) {}
    function revokeAccessToAssetId(string memory _assetId, address _entity) public onlyPatient {}
    function viewMedicalAssetId(address _patient, string memory _assetId)
        public view onlyAuthorizedForAssetId(_patient, _assetId) returns (string memory) {}
}
```

Figure 16: Asset Management Smart Contract

4.6 DATA STRUCTURES AND STATE VARIABLES

The contract employs a `Patient` struct that encapsulates all relevant patient information. This struct contains:

- **`patientAddress`** (type `address`): Uniquely identifies the patient within the Ethereum network. The address type is a 20-byte value used to identify both externally owned accounts (EOA) and contract accounts.
- **`medicalAssetIds`** (type `array of strings`): Maintains a list of identifiers for the patient's medical assets. Each entry in this dynamically sized array represents a unique asset identifier.

- **`accessGrantedPerAssetId`** (type `nested mapping`): Maps asset IDs to another mapping that links addresses to a boolean value, indicating authorization to access the specified medical asset.

Furthermore, the contract declares two critical state variables:

- **`patients`** (type `mapping`): Links Ethereum addresses to `Patient` records, facilitating quick data retrieval based on Ethereum addresses.
- **`patientAddresses`** (type *array*): Stores the addresses of all registered patients, enabling iteration over all patient data.

4.7 ACCESS CONTROL MODIFIERS

Modifiers in Solidity are code that can be run before and/or after a function call to modify its behavior. They are used to simplify code and enhance security, which in this case is used to register custom validators that grant or deny access to the user trying to execute this contract's code. Since a few functions must only allow the owner to execute some querying upon some piece of information, two modifiers were introduced:

1. **`onlyPatient`**: Ensures that only the patient whose data is being accessed or modified can execute specific functions.
2. **`onlyAuthorizedForAssetId`**: Allows function execution not only by the patient but also by others who have been granted explicit access to specific medical assets.

4.8 FUNCTIONS

In Solidity a function is defined by using the `function` keyword, followed by the name of the function, which is unique and does not match with any of the reserved keywords. A function can also have a list of parameters containing the name and data type of the parameter. The return value of a function is optional but in solidity, the return type of the function, is defined at the time of declaration.

4.8.1 *registerPatient Function*

The `registerPatient` function, shown in Algorithm 1, enables the registration of a new patient. It employs a check to ensure that the address being registered does not already correspond to a registered patient. It uses uses the patient's address initialization state to confirm whether the mapping already has a record for that address. Upon successful registration, the function updates the patients mapping and appends the new patient's address to the `patientAddresses` array, broadcasting a `PatientRegistered` event to signal the change.

Algorithm 1 Register Patient

```
1: function RegisterPatient(patient)
2:     if patients[patient].patientAddress ≠ address(0) then
3:         throw "Patient already registered"
4:     end if
5:     patients[patient].patientAddress ← patient
6:     patientAddresses.append(patient)
7:     emit PatientRegistered(msg.sender, patient)
8: end function
```

This pseudocode checks if the patient is already registered by looking up their address in a mapping. If not found, it sets their address in the system, appends it to a list of patient addresses, and emits a registration event.

4.8.2 *addMedicalAssetId Function*

The `addMedicalAssetId` function associates a new medical asset ID with a patient's record. The algorithm is outlined in 2.

Algorithm 2 Add Medical Asset ID

```
1: function AddMedicalAssetId(assetId, patient)
2:     patients[patient].medicalAssetIds.push(assetId)
3: end function
```

It allows medical asset IDs to be associated with a specific patient. This function lacks access controls, in other words, does not implements any of the modifiers discussed above.

4.8.3 *getAllAssetIds Function*

The `getAllAssetIds` function (see in Algorithm 3) is a view function, meaning it does not alter the state on the blockchain but returns data.

Algorithm 3 Get All Asset IDs

```
1: function GetAllAssetIds
2:     allAssetIds ← new array of size patientAddresses.length
3:     for i ← 0 to patientAddresses.length - 1 do
4:         allAssetIds[i] ← patients[patientAddresses[i].medicalAssetIds
5:     end for
6:     return allAssetIds
7: end function
```

It constructs a two-dimensional array where each sub-array contains the medical asset IDs associated with a specific patient. This is used to fetch a comprehensive list of all assets for every patient, valuable for audits or interfaces that require display of all patient assets. This pseudocode demonstrates how the function initializes an array sized by the number of registered patients and iteratively fetches and stores medical asset IDs from each patient's record into this array, eventually returning it.

4.8.4 *grantAccessToAssetId Function*

The `grantAccessToAssetId` function, as shown in Algorithm 4, allows a patient to grant access to a specific medical asset. The function updates the `accessGranted PerAssetId` mapping within the patient's record, linking the provided asset ID with the entity's address to grant access. This function is protected by the `onlyPatient` modifier, ensuring that only the patient can manage access permissions for their medical assets.

Algorithm 4 Grant Access to Asset ID

```
1: function GrantAccessToAssetId(assetId, entity)
2:     patients[msg.sender].accessGrantedPerAssetId[assetId][entity] ← true
3: end function
```

4.8.5 *viewAllAssetIds Function*

The `viewAllAssetIds` function, detailed in Algorithm 5, enables a patient to view all their associated medical asset IDs. It retrieves and returns the array of medical asset IDs stored within the patient's record. This function is guarded by the `onlyPatient` modifier to ensure that only the patient can access their list of medical assets.

Algorithm 5 View All Asset IDs

```
1: function ViewAllAssetIds
2:     return patients[msg.sender].medicalAssetIds
3: end function
```

4.8.6 *revokeAccessToAssetId Function*

The `revokeAccessToAssetId` function, shown in Algorithm 6, allows a patient to revoke access to a specific medical asset from an entity. It updates the `accessGrantedPerAssetId` mapping, setting the access flag to `false` for the specified asset ID and entity address. This action is similarly restricted to the patient through the `onlyPatient` modifier, ensuring that only the patient can manage their access permissions.

Algorithm 6 Revoke Access to Asset ID

```
1: function RevokeAccessToAssetId(assetId, entity)
2:     patients[msg.sender].accessGrantedPerAssetId[assetId][entity] ← false
3: end function
```

4.8.7 *viewMedicalAssetId Function*

The `viewMedicalAssetId` function, detailed in Algorithm 7, allows authorized entities to view a specific medical asset ID associated with a patient. The function first verifies whether the asset ID exists within the patient's medical assets by comparing the hashed values of the asset ID with each entry in the patient's list. It requires that the entity is authorized to view the asset, enforced by the `onlyAuthorizedForAssetId` modifier. If the asset exists, and the entity is authorized, the function returns the asset ID.

Algorithm 7 View Medical Asset ID

```
1: function ViewMedicalAssetId(patient, assetId)
2:     assetExists ← false
3:     for each asset in patients[patient].medicalAssetIds do
4:         if keccak256(bytes(asset))  keccak256(bytes(assetId)) then
5:             assetExists ← true
6:             break
7:         end if
8:     end for
9:     require assetExists, "Asset ID does not exist"
10:    return assetId
11: end function
```

4.9 IDENTITY MANAGEMENT USING ZERO-KNOWLEDGE PROOFS

Before entering the core implementation of the identity management mechanism some aspects about how zkSNARK works must be explained. These types of Zero-Knowledge Proofs rely on Elliptic Curves (EC) [82] rather than Public-Key Cryptosystems, or, more commonly known as, RSA (Rivest-Shamir-Adleman) algorithms [39]. Instead of relying on prime numbers like RSA does, EC are structured algebraically over finite fields, therefore increasing the difficulty to decrypt them mathematically [83]. Moreover, elliptic curves not only provide higher security cover and a smaller sized algorithm with rapid execution times, a major advantage when used in heavy environments like a ledger. The Identity Management Smart contract, or more concisely - `Verifier.sol`, involves elliptic curve operations in groups G1 and G2. Here, G1 is a group on the elliptic curve over a prime field, and G2 is a group on a twist of the curve over an extension field. The operations on these groups underpin the zero-knowledge proofs by facilitating complex mathematical relations securely and efficiently.

This section delineates the utilization of elliptic curve cryptography (ECC) in the smart contract, elucidating the mathematical functions and their direct implications in the cryptographic verification process, especially in the context of zkSNARK on Ethereum.

4.10 ZKSNARK CODE OVERVIEW

The ZoKrates code presented in Listing 1 implements a function that securely computes a SHA-256 hash based on four private keys.Two from the patient and two

from the healthcare provider. Also, it verifies that the generated hash matches the corresponding public hash parts provided as input.

```
1  from "hashes/sha256/512bitPacked" import main as sha256packed;
2
3  def main(private field patientPrivateKey0, private field
   ↪  patientPrivateKey1, private field healthcarePrivateKey0,
   ↪  private field healthcarePrivateKey1, field hashPart0, field
   ↪  hashPart1) {
4      field[2] hash = sha256packed([patientPrivateKey0,
       ↪  patientPrivateKey1, healthcarePrivateKey0,
       ↪  healthcarePrivateKey1]);
5      assert(hash[0] == hashPart0);
6      assert(hash[1] == hashPart1);
7      return;
8  }
```

Listing 1: Compute function algorithm - *ZoKrates*

As referenced in Listing 1, the `sha256/512bitPacked` module contains an optimized implementation of the SHA-256 hashing algorithm for use in ZoKrates, allowing the function to handle 512-bit inputs efficiently despite the field size limitations inherent in ZoKrates.

The core of the code is encapsulated within the `main` function, which accepts six parameters. These include `patientPrivateKey0` and `patientPrivateKey1`, representing parts of the patient's private key, and `healthcarePrivateKey0` and `healthcarePrivateKey1`, representing parts of the healthcare provider's private key. Additionally, the function takes `hashPart0` and `hashPart1`, which are public fields representing parts of an expected hash value. The distinction between private and public fields is crucial, as private fields are inputs that the Prover wishes to keep confidential, while public fields are inputs that can be shared with the Verifier.

Within the function, the SHA-256 hash of the concatenated private key parts is computed using the `sha256packed` function. This function takes an array of four 128-bit field elements as input, representing the concatenated private keys. Given the limitations of ZoKrates, where a field value can only hold 254 bits, the function handles the 512-bit input by splitting it into four 128-bit segments. The result is a `field` array of size 2, named `hash`, which contains the two 256-bit parts of the SHA-256 hash. This step illustrates how private inputs are transformed into a hashed value that can be verified against public parameters.

The function includes two assertions (line 5 and 6 of Listing 1) to verify that the computed hash parts (`hash[0]` and `hash[1]`) match the provided expected hash parts (`hashPart0` and `hashPart1`). Assertions are fundamental in zkSNARK,

forming the basis of the proof. If these assertions hold true, it implies that the Prover knows the private key parts that produce the given hash, thus validating the integrity and authenticity of the private keys without disclosing them. The function ends with a return statement, which in this context does not return any value but signifies the end of the function. The primary purpose of the function is to validate the assertions, thereby proving the knowledge of the inputs without revealing them.

As a result of this function, ZoKrates creates a file, `proof.json`, consisting of the three elliptic curve points that make up the proof. This can be used by the Verifier to validate the assertions made in the program.

4.11 VERIFIER CONTRACT

The 'Verifier' contract is responsible for verifying zkSNARK proofs on the blockchain. It includes structures for the verifying key and the proof, as well as functions to set up these structures and verify proofs.

```
Struct VerifyingKey:
        Pairing.G1Point alpha;
        Pairing.G2Point beta;
        Pairing.G2Point gamma;
        Pairing.G2Point delta;
        Pairing.G1Point[] gamma_abc;
```

Listing 2: VerifyingKey Struct

The `VerifyingKey` struct, defined on Listing 2, contains elliptic curve points $(\alpha, \beta, \gamma, \delta)$ required for zkSNARK proof verification. These points are precomputed during the zkSNARK setup phase. These points are part of the zkSNARK public parameters used in pairing checks to validate the proof.

The γ_abc array contains points used in a linear combination with the public input to form part of the proof verification.

4.11.1 *Correlation with proposed solution*

In the ZoKrates code (see Listing 1), the `private` fields (`patientPrivateKey0`, `patientPrivateKey1`, `healthcarePrivateKey0`, `healthcarePrivateKey1`) are the private inputs, while `hashPart0` and `hashPart1` are the public inputs. The `verifyingKey`

function in the `Verifier` contract initializes the verifying key with these precomputed elliptic curve points.

4.11.2 *Proof Structure*

The `Proof` struct (Listing 3) contains the zkSNARK proof, consisting of elliptic curve points `a`, `b`, and `c`.

```
Struct Proof:
        Pairing.G1Point ba
        Pairing.G2Point b;
        Pairing.G1Point c;
        Pairing.G2Point delta;
        Pairing.G1Point[] gamma_abc;
```

Listing 3: Proof Struct

These points are generated during the proof creation process and are used in the verification process.

4.11.3 *Setup Verifying Key*

The `verifyingKey` algorithm, initializes the verifying key with precomputed elliptic curve points, as can be seen in Algorithm 8. The values for $\alpha, \beta, \gamma, \delta$ are derived from the public parameters generated during the zkSNARK setup phase. The γ_abc represents the linear combination of the public inputs (`hashPart0` and `hashPart1`) and is used to form the proof verification equation.

Algorithm 8 Setup Verifying Key

Require: None
Ensure: Verifying key vk
1: Initialize empty verifying key structure vk
2: vk.α ← Pairing.G1Point(uint256 value 1, uint256 value 2)
3: vk.β ← Pairing.G2Point(
    [uint256 value 3, uint256 value 4], [uint256 value 5, uint256 value 6])
4: vk.γ_abc ← Pairing.G2Point(
    [uint256 value 7, uint256 value 8], [uint256 value 9, uint256 value 10])
5: vk.δ ← Pairing.G2Point(
    [uint256 value 11, uint256 value 12], [uint256 value 13, uint256 value 14])
6: vk.γ_abc[0] ← Pairing.G1Point(uint256 value 15, uint256 value 16)
7: vk.γ_abc[1] ← Pairing.G1Point(uint256 value 17, uint256 value 18)
8: vk.γ_abc[2] ← Pairing.G1Point(uint256 value 19, uint256 value 20)
9: **return** vk

4.11.4 *Verify Function*

On Algorithm 9 we can see the design of the core proof validation algorithm. The algorithm begins by defining the `snark_scalar_field`, which ensures that inputs are within the correct field size for the elliptic curve operations. The verifying key is initialized using the `verifyingKey` function. Afterwards, the function checks that the length of the input plus one matches the length of `vk.`γ`_abc`. This ensures that the input is correctly formatted and matches the expected structure of the zkSNARK setup. The function computes a linear combination of the `vk.`γ`_abc` points using the input values. This involves scaling each point in `vk.`γ`_abc` by the corresponding input value and summing the results to form a new elliptic curve point `vk_x`.

Algorithm 9 Verify zkSNARK Proof

Require: Input array input, Proof structure proof
Ensure: Verification result (0 for success, 1 for failure)
Define snark_scalar_field as 2.19×10^{76}
▷ Full value:
▷ 21888242871839275222246405745257275088
▷ 548364400416034343698204186575808495617
Initialize VerifyingKey vk by calling VerifyingKey()
Assert that input.length + 1 equals vk.γ_*abc*.length
Initialize G1Point vk_x as (0, 0)
for each element i in input **do**
    Assert that input[i] < snark_scalar_field
    vk_x ← vk_x + scalar_mul(vk.γ_*abc*[i + 1], input[i])
end for
vk_x ← vk_x + vk.γ_*abc*[0]
result ← pairingProd4(
    proof.a, proof.b,
    negate(vk_x), vk.γ,
    negate(proof.c), vk.δ,
    negate(vk.α), vk.β)
if result is false **then**
    return 1
else
    return 0
end if

Each point in `vk.`γ`_abc` is multiplied by the corresponding input value. This operation involves repeated addition of the point and is crucial for encoding the input values into the elliptic curve operations. The results of the scalar multiplications are added together to form the linear combination. Point addition on elliptic curves combines two points to produce a third point that lies on the curve. The function performs a pairing check using the proof points (`a`, `b`, `c`), the linear combination point (`vk_x`), and the precomputed verifying key points ($\alpha, \beta, \gamma, \delta$). The pairing check ensures that the elliptic curve points interact correctly and that the proof is valid. If the pairing product equals 1, the proof is considered valid; otherwise, it is invalid.

4.11.5 *VerifyTx Function*

The `Verify Transaction` (Algorithm 10) provides a public interface for proof verification. The algorithm converts the input array into a format suitable for the `verify` function. This ensures that the input values are correctly formatted and prepared for verification. The algorithm then calls the `Verify` function with the converted input values and the proof. If the `Verify` function returns 0, the proof is valid, and the function returns true. Otherwise, it returns false, indicating an invalid proof.

Algorithm 10 Verify Transaction

Require: Proof structure proof, Array input of length 2
Ensure: Boolean verification result
1: Initialize empty array inputValues
2: **for** each element i in input **do**
3: Append input[i] to inputValues
4: **end for**
5: result ← Verify(inputValues, proof)
6: **if** result equals 0 **then**
return true
7: **else**
return false
8: **end if**

4.12 ELLIPTIC CURVE POINTS AND PAIRING OPERATIONS

Elliptic curve points are elements on an elliptic curve, defined by a specific mathematical equation. In the context of zkSNARK, these points are used to encode cryptographic information securely. Points in G1 and G2 (see Section 4.11.2) are used to construct proofs, and pairing operations verify the correctness of these proofs. Pairing operations map pairs of points from G1 and G2 to an element in a target group. This operation ensures that the elliptic curve points interact correctly under cryptographic rules, which is essential for zkSNARK proof verification.

In the verification process, specific points known as α, β, γ, and δ are used. These points are part of the verifying key, a crucial component in the zkSNARK setup that ensures the validity of proofs. These points are public parameters used in the pairing checks to verify the proof's validity. The array of these points: γ_{abc}, incorporates the public input into the verification process, ensuring that the proof corresponds to the given input. Each point in γ_{abc} represents a component of the public input, scaled appropriately and combined in the verification equation.

4.12.1 *Generating the Base Points*

The base points (generators) for G1 and G2 are defined in Algorithms 11 and 12.

Algorithm 11 Generator of G1

Ensure: G1Point G_1
1: **return** G1Point(1,2)

Algorithm 12 Generator of G2

Ensure: G2Point G_2
1: $x_1 \leftarrow 1.09 \times 10^{76}$
2: $x_2 \leftarrow 1.16 \times 10^{76}$
3: $y_1 \leftarrow 8.50 \times 10^{75}$
4: $y_2 \leftarrow 4.08 \times 10^{75}$
5: **return** G2Point(x_1, x_2, y_1, y_2) ▷ Full values in A

G1POINT: Represents a point on the elliptic curve G1. The generator (base point) of G1 is set to (1, 2). The generator point of G1 (the subgroup of the elliptic curve over a prime field) is often a simple, low-value point. The point (1,2) may be used as a generator for demonstration or simplicity, if it indeed belongs to the curve and generates a large enough subgroup (typically, the size of the subgroup should be a large prime or have a large prime factor to ensure security). For certain elliptic curves used in zkSNARK (like the BN256 curve often used with Ethereum [84]), the generator points are precomputed and hardcoded to ensure they are secure and efficient for cryptographic operations.

G2POINT: Represents a point on the elliptic curve G2. The generator of G2 is a more complex point defined by two coordinates, each being an array of two `uint256` values. The coordinates of G2 (which lives in a quadratic extension of the base field, hence the coordinates being arrays of two numbers) are far less trivial to determine. They require careful selection to ensure they have the correct order (number of elements in the subgroup they generate) and work correctly with the bilinear pairing operation. So, the hardcoded numbers used in the algorithm are chosen to meet the security requirements (like being a part of a subgroup of prime order) and optimize the pairing computation. These values are not arbitrary; they ensure that the cryptographic protocols leveraging these points (e.g., zkSNARK) are both secure and efficient. These values are predefined by the cryptographic protocol being used, in this case, the G2 generator needs to be part of a specific coset of the curve to properly interact with G1 under the pairing [39].

```
        assembly {
                success := staticcall(sub(gas(), 2000), 6, input, 0xc0, r, 0x60)
                switch success case 0 { invalid() }
        }
```

Listing 4: Low-level implementation of EllipticCurveAdd in Solidity assembly

4.12.2 *Point Negation*

Negation of a point on an elliptic curve involves inverting the y-coordinate. For a point $p\ \ X, Y$, the negation is $-p\ \ X, -Y$, as defined in Algorithm 13.

Algorithm 13 Negation of G1 Point

Require: G1Point p
Ensure: Negated G1Point
1: Define q as 21.9×10^{76} ▷ Full value in A
2: **if** p.X = 0 and p.Y = 0 **then**
    return G1Point(0, 0)
3: **end if**
4: negY ← q - (p.Y mod q) **return** G1Point(p.X, negY)

PRIME Q: Represents the order of the base field F_q for G1.

NEGATION OPERATION: Inverts the y-coordinate modulo q, effectively reflecting the point across the x-axis.

4.12.3 *Point Addition*

Adding two points on an elliptic curve involves algebraic operations that result in a third point on the curve (Algorithm 14).

Algorithm 14 Addition of G1 Points

Require: G1Point p1, G1Point p2
Ensure: G1Point r (sum of p1 and p2)
1: Initialize array input of size 4
2: $input_0 \leftarrow p_1.X$
3: $input_1 \leftarrow p_1.Y$
4: $input_2 \leftarrow p_2.X$
5: $input_3 \leftarrow p_2.Y$
6: success, r ← EllipticCurveAdd(input)
7: **if** not success **then**
    Raise ErrorException
8: **end if**
    return r
▷ EllipticCurveAdd is a low-level function for efficient elliptic curve point addition detailed on Listing 4

STATICCALL [85]: Executes the precompiled elliptic curve addition function. The inputs are coordinates of the two points, and the result is stored in `r` .

```
        assembly {
                success := staticcall(sub(gas(), 2000), 7, input, 0x80, r, 0x60)
                switch success case 0 { invalid() }
        }
```

Listing 5: Low-level implementation of EllipticCurveScalarMul in Solidity assembly

4.12.4 *Scalar Multiplication*

Scalar multiplication involves multiplying a point on the elliptic curve by a scalar value, which is crucial for elliptic curve cryptography, as can be seen in Algorithm 15.

Algorithm 15 Scalar Multiplication of G1 Point

Require: G1Point p, scalar s
Ensure: G1Point r (result of scalar multiplication)
Initialize array input of size 3
$input_0 \leftarrow p.X$
$input_1 \leftarrow p.Y$
$input_2 \leftarrow s$
success, r ← EllipticCurveScalarMul(input)
if not success **then**
    Raise ErrorException
end if
return r
▷ EllipticCurveScalarMul is a low-level function for efficient elliptic curve scalar multiplication detailed on Listing 5

4.12.5 *Pairing Check*

The pairing check involves verifying that a certain relation holds between multiple pairs of points from G1 and G2.

Algorithm 16 Pairing Check

Require: Arrays of G1Points p1 and G2Points p2
Ensure: Boolean result of pairing check
Assert p1.length equals p2.length
elements ← p1.length
inputSize ← elements × 6
Initialize input as array of size inputSize
for i from 0 to elements - 1 **do**
    $input_{i \times 6\ 0} \leftarrow p1i.X$
    $input_{i \times 6\ 1} \leftarrow p1i.Y$
    $input_{i \times 6\ 2} \leftarrow p2i.X_1$
    $input_{i \times 6\ 3} \leftarrow p2i.X_0$
    $input_{i \times 6\ 4} \leftarrow p2i.Y_1$
    $input_{i \times 6\ 5} \leftarrow p2i.Y_0$
end for
success, result ← PairingOperation(input)
if not success **then**
    Raise ErrorException
end if
return $result \leftarrow 0$
▷ PairingOperation is a low-level function for efficient elliptic curve pairing checks (Listing 6

```
        assembly {
                success := staticcall(sub(gas(), 2000), 8, add(input, 0x20), mul(inputSize,
                ↪  0x20), out, 0x20)
                switch success case 0 { invalid() }
        }
```

Listing 6: Low-level implementation of PairingOperation in Solidity assembly

INPUT PREPARATION : Constructs an input array for the pairing precompiled function, interleaving the coordinates of points from G1 and G2.

STATICCALL: Executes the pairing check, which verifies that the elliptic curve points satisfy the bilinear pairing equation.

4.12.6 *Convenience Pairing Functions*

Algorithms 17, 18 and 19 provide simplified interfaces for performing pairing checks with 2, 3, or 4 pairs of points.

Algorithm 17 Pairing Check for Two Pairs

Require: G1Points a1 and b1, G2Points a2 and b2
Ensure: Boolean result of pairing product
Initialize array p1 of G1Points with size 2
Initialize array p2 of G2Points with size 2
$p_{10} \leftarrow a_1$
$p_{11} \leftarrow b_1$
$p_{20} \leftarrow a_2$
$p_{21} \leftarrow b_2$
result ← PairingCheck(p1, p2) ▷ See Algorithm 16
return result

Algorithm 18 Pairing Check for Three Pairs

Require: G1Points a1, b1, and c1, G2Points a2, b2, and c2
Ensure: Boolean result of pairing product
Initialize array p1 of G1Points with size 3
Initialize array p2 of G2Points with size 3
$p_{10} \leftarrow a_1$
$p_{11} \leftarrow b_1$
$p_{12} \leftarrow c_1$
$p_{20} \leftarrow a_2$
$p_{21} \leftarrow b_2$
$p_{22} \leftarrow c_2$
result ← PairingCheck(p1, p2) ▷ See Algorithm 16
return result

These algorithms reduce the complexity of performing pairing checks by allowing multiple pairs of points to be verified simultaneously. They create arrays of G1 and G2 points and call the `pairing` function to perform the actual check.

Algorithm 19 Pairing Check for Four Pairs

Require: G1Points a1, b1, c1, and d1, G2Points a2, b2, c2, and d2
Ensure: Boolean result of pairing product
Initialize array p1 of G1Points with size 4
Initialize array p2 of G2Points with size 4
$p_{10} \leftarrow a_1$
$p_{11} \leftarrow b_1$
$p_{12} \leftarrow c_1$
$p_{13} \leftarrow d_1$
$p_{20} \leftarrow a_2$
$p_{21} \leftarrow b_2$
$p_{22} \leftarrow c_2$
$p_{23} \leftarrow d_2$
result ← PairingCheck(p1, p2) ▷ See Algorithm 16
return result

4.13 GENERATING PROOFS AND COMPUTING WITNESSES

The earlier subsections described the backbone of the logic behind the ZoKrates. To further understand how Zero-Knowledge Proofs may be applied to a real scenario we will utilize the ZoKrates algorithm and provide real values. The following scenario will take into consideration a Patient X and an Healthcare Provider Y. Patient X will try to prove that he withholds a proof that corroborates there is in fact a relationship between him and Healthcare Provider Y. Assuming Patient X already has registered himself in the platform and correctly onboarded by performing a Data Sharing Agreement with Healthcare Y, it would be correct to assume that every entity has their own private keys regarding their smart accounts. As seen in the ZoKrates algorithm 1, the main function receives two private fields for each Patient X and Healthcare Y, as well a public key, split in two parts. This public key will be referred here as a "Shared Key" as both parties know its value. To replicate these 256 bit keys a JavaScript program was implemented.

This section of the document breaks down the JavaScript program in smaller chunks and explains the process of generating Private and Shared Keys using cryptographic functions. These keys are used in the ZoKrates function to ensure secure and privacy-preserving operations. The code uses the Node.js `crypto` module to perform HMAC (Hash-based Message Authentication Code) and SHA-256 (Secure Hashing Algorithm) hashing operations.

The provided code generates private keys, for a patient and a healthcare provider, from a master key and unique identifiers. It then derives a shared key from these private keys. This shared key can be used in the ZoKrates function to verify computations securely.

Hash-based Message Authentication Code (HMAC) is a cryptographic technique that combines a cryptographic hash function with a secret key [86]. In this context, HMAC is used to derive unique keys based on a master key and unique identifiers.

4.13.1 *Derive Key Function*

The `deriveKey` function, present in Listing 7, generates a private key from a master key and a unique identifier, using HMAC with SHA-256.

```
const crypto = require('crypto');

function deriveKey(masterKey, uniqueId) {
        const hmac = crypto.createHmac('sha256', masterKey);
        hmac.update(uniqueId);
        const derivedKey = hmac.digest();
        const derivedKeyParts = [
        BigInt('0x' + derivedKey.slice(0, 16).toString('hex')),
        BigInt('0x' + derivedKey.slice(16).toString('hex'))
        ];

        return derivedKeyParts.map(part => part.toString());
}
```

Listing 7: Derive Key Function

The derive key function employs a series of cryptographic operations to generate a unique key from a master key and a unique identifier. The process begins with the creation of an HMAC using the SHA-256 algorithm, initialized with the provided master key. This HMAC is then updated with the unique identifier, ensuring the derived key is specific to this particular combination of master key and identifier. Once updated, the HMAC digest is computed, yielding a 32-byte buffer that serves as the raw derived key material. This buffer is then split into two 16-byte parts, each of which is converted into a BigInt and subsequently into a hexadecimal string. The function ultimately returns these two parts as an array of strings, each representing half of the derived key in hexadecimal format. This approach ensures a consistent and secure method of key derivation, suitable for various cryptographic applications.

4.13.2 *Derive Shared Key Function*

The `deriveSharedKey`, present in Listing 8, function generates a shared key from two private keys using SHA-256 hashing.

The derive `derivedSharedKey` function implements a process that generates a *Shared Key* from two input keys. It begins by concatenating the hexadecimal

```
const crypto = require('crypto');

function deriveSharedKey(key1, key2) {
        const concatenatedKeys = Buffer.concat([
        Buffer.from(BigInt(key1[0]).toString(16).padStart(32, '0'), 'hex'),
        Buffer.from(BigInt(key1[1]).toString(16).padStart(32, '0'), 'hex'),
        Buffer.from(BigInt(key2[0]).toString(16).padStart(32, '0'), 'hex'),
        Buffer.from(BigInt(key2[1]).toString(16).padStart(32, '0'), 'hex')
        ]);

        const hash = crypto.createHash('sha256');
        hash.update(concatenatedKeys);
        const sharedKey = hash.digest();

        const sharedKeyParts = [
        BigInt('0x' + sharedKey.slice(0, 16).toString('hex')),
        BigInt('0x' + sharedKey.slice(16).toString('hex'))
        ];

        return sharedKeyParts.map(part => part.toString());
}
```

Listing 8: Derive Shared Key Function

representations of both parts of `key1` and `key2`, creating a single, comprehensive string of key material. This concatenated string is then fed into a SHA-256 hash function, which processes the input to produce a cryptographically secure hash. The resulting 32-byte buffer represents the raw *Shared Key*. To format this for use, the function splits the hash into two equal 16-byte parts. Each part is then converted into a `BigInt` and subsequently into a hexadecimal string, ensuring a standardized representation. The function ends by returning these two parts as an array of strings, with each string representing half of the *Shared Key* in hexadecimal format. This method ensures a consistent and secure way of deriving a *Shared Key* from two input keys, suitable for various cryptographic protocols and secure communication scenarios.

4.14 KEY DERIVATION AND SHARED KEY GENERATION

The code on Listing 9 demonstrates the derivation of private keys for a patient and a healthcare provider, as well as the generation of a *Shared Key*.

The key derivation process begins with a securely generated *Master Key*, which serves as the foundation for all subsequent key generation. This process utilizes unique identifiers, for both the patient and the healthcare provider, ensuring the specificity of the derived keys. The system then employs the `deriveKey` function (see Listing 7), leveraging the *Master Key* in combination with these unique identifiers to generate distinct private keys for both the patient and the healthcare provider. These derived private keys are then used as inputs for the `deriveSharedKey` function (see

```
const masterKey = 'securely generated master key';
const patientId = 'patient id';
const healthcareId = 'healthcare id';

const patientPrivateKey = deriveKey(masterKey, patientId);
const healthcarePrivateKey = deriveKey(masterKey, healthcareId);

const sharedKey = deriveSharedKey(patientPrivateKey, healthcarePrivateKey);

console.log('Patient Private Key: ', patientPrivateKey);
console.log('Healthcare Private Key: ', healthcarePrivateKey);
console.log('Shared Key: ', sharedKey);
```

Listing 9: Key Derivation Example

Listing 8), which computes a *Shared Key* that can be used for secure communication between the patient and healthcare provider. This *Shared Key* represents a cryptographic link between the two parties, derived from their individual private keys. Finally, the system outputs both the derived private keys and the resulting *Shared Key* to the console, providing a transparent view of the key generation process.

This methodology ensures a secure, deterministic, and verifiable approach to key generation and sharing in healthcare information systems.

4.15 FRONTEND APPLICATION

Addressing missing User Interface and Experience (UI/UX) problems with the use of Account Abstraction is better exposed with practical examples. For this exact reason a PoC for patient registration was implemented. This Web application acts as the facade for the registration in the system's platform. It allows the user to create a Smart Account using Magic Signer backed with Alchemy's Account Kit. Integrating these tools in a frontend application can be challenging, thus, the development platform chosen was NextJS [87]. It's based on React, a well-know JavaScript library that provides numerous features and support from a large community of developers. NextJS facilitates the decoupling from every module used in the Web Application, therefore allowing posterior changes, such as: using a different Signer or another implementation of a Smart Wallet.

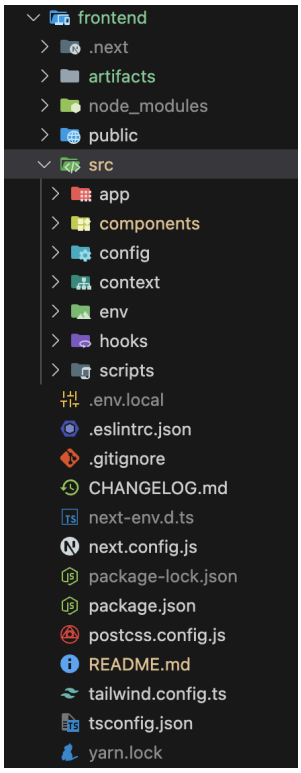


Figure 17: Frontend directory structure

This section won't dive into much detail on how a NextJS application works, the relevant key points are how the Magic Signer and Alchemy's AccountKit are integrated.

4.16 CONFIGURING ALCHEMY

To setup Alchemy as the default provider of this application and the API that connects the application to the Ethereum Test Network it is necessary to create a new application in Alchemy's Dashboard[8]: (see Figure 19 and 20)

8 https://dashboard.alchemy.com/apps

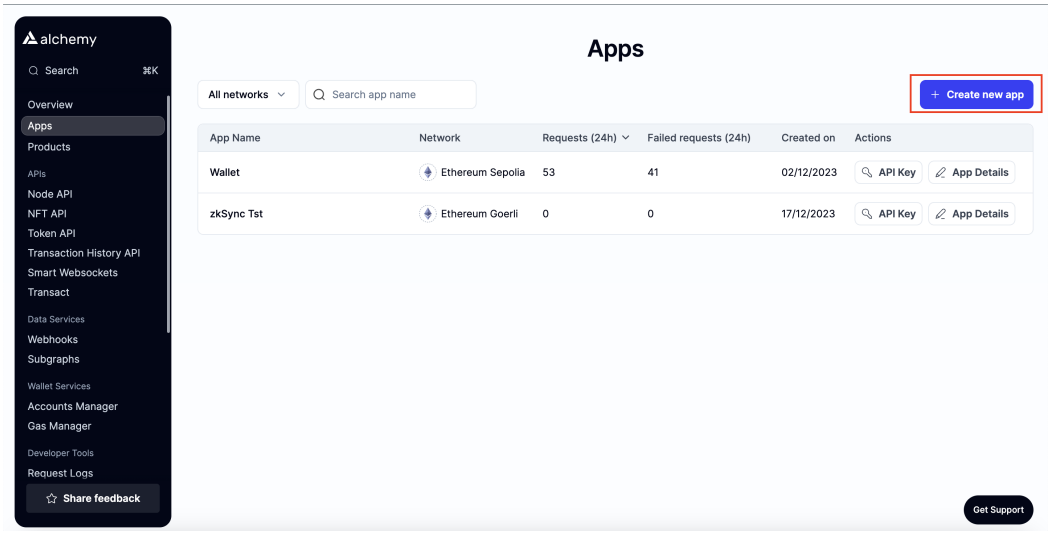


Figure 18: Alchemy's Dashboard

By pressing "Create new app" button a modal pop's up:

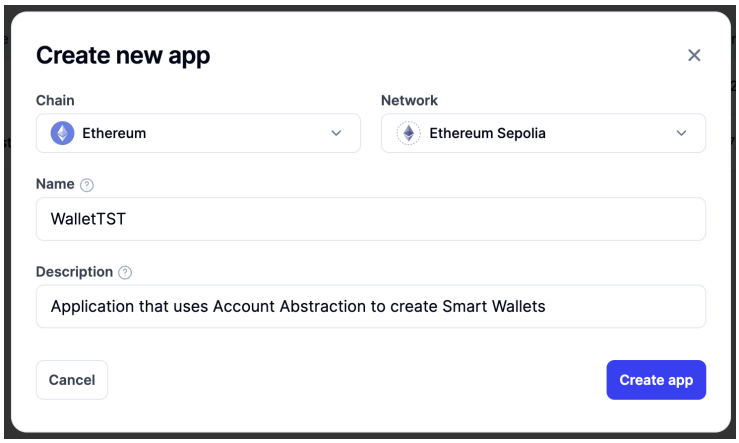


Figure 19: Alchemy's Create New App Modal

The most important aspect is to change the Network to "Ethereum Sepolia". Otherwise the application won't work correctly and cannot connect to Smart Contracts that are deployed via Hardhat on the Sepolia Test Net.

"Apps" (in Figure 18) are just a project configuration that allows the developer to namespace applications. The main purpose here is to get a dedicated API Key to use Alchemy's feature such as: connecting to the Sepolia network. This way, this Web application will be coupled to this API Key. Pressing the "API Key" button in the actions column will reveal the necessary information to connect the Web Application to the Alchemy Provider:

Figure 20: Alchemy's Credentails Details

In Figure 20, the first two fields are the most relevant ones. The first is the secret API Key for the application and the second one is the RPC URL used to communicate to the blockchain and send requests that Alchemy interpretates and transpile them as blockchain queries.

These configuration tokens can be used by injecting them in a local environment file: `env.local`. This file is represented by figure 21.

```
ALCHEMY_KEY=eIfwt8ic7hz**********
NEXT_PUBLIC_MAGIC_API_KEY=pk_live_00742C********
ALCHEMY_RPC_URL=https://eth-sepolia.g.alchemy.com/v2/<ALCHEMY_KEY>
NEXT_PUBLIC_ALCHEMY_GAS_MANAGER_POLICY_ID=**af8008-****-4857-****-8ea658b714b1
```

Figure 21: Frontend Environment Variables

Notice that variables: `NEXT_PUBLIC_MAGIC_API_KEY` and `NEXT_PUBLIC_ALCHEMY_GAS_MANAGER_POLICY_ID` are exposed in this file. The first one is for Magic Signer, which will be approached in the next section. The second one is important for the context of this project because it allows to sponsor gas fees for patients. Alchemy allows to configure a Gas Policy[9] which defines the spending limit. It represents the amount of money spent per User Operation or the amount of User Operations that can be sponsored. This policy also allows to grant or revoke the sponsorship per Wallet address. For this configuration, Alchemy's dashboard has a dedicated section called "Gas Manager", which is represented in Figure 22.

9 https://docs.alchemy.com/docs/setup-a-gas-manager-policy

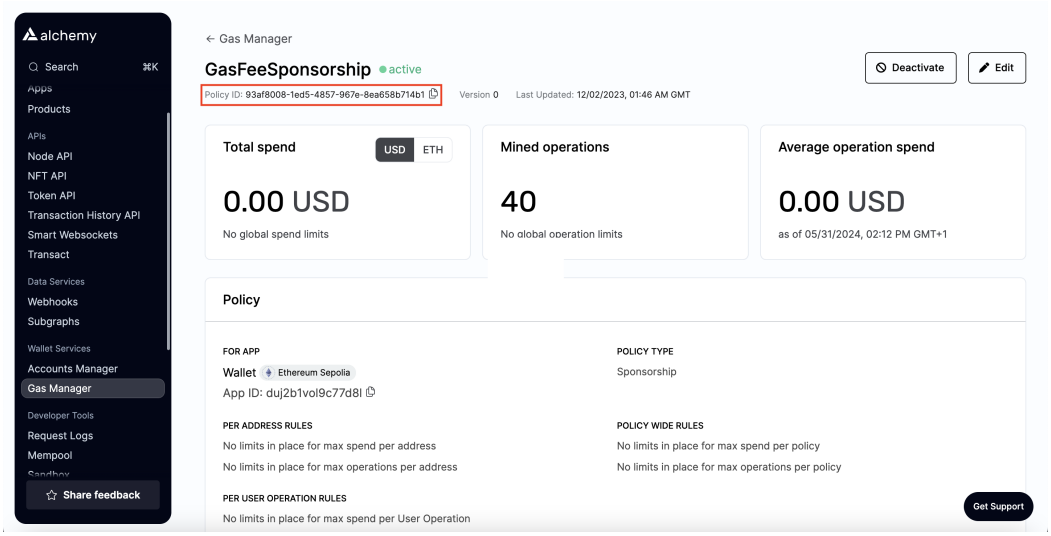


Figure 22: Gas Manager API

After creating a Policy with the suggested default values for each parameter the Gas Policy is active and returns a Policy ID as seen in Figure 22. This ID is then used in the `.env.local` file, mentioned above.

In the real scenario, the entity that implements this mechanism is going to sponsor the transaction fees (gas) per user. Patients wouldn't want to use a system that charges them for each exam made and stored in their Patient Wallet. Although, a business policy can define this in the contract between the patient and the provider of this service. For example, the medical provider can charge patients a monthly subscription that accounts for all these fees and usage charges. Nevertheless, this is out-of-scope for this project.

4.17 CONFIGURING A SIGNER

Alchemy's AccountKit allows ease of integration with multiple *Signers*. Currently the most suited *Signer* is Magic [67], as it offers all the necessary features for implementing a solution for the problems addressed in this thesis. An important aspect consider is the type of *Signer* that Magic is. It ensures a safe place to store assets (Wallet) and only the user can access it via password or any other verification method. This is known as Non-Custodial Wallet. Basically, this implementation delegates the infrastructure and management of how the Wallet's private key is stored to the third party *Signer* without sacrificing security. Combining this with social logins, the patient/user can register themselves through their social media

account, such as their Google Account, and use those credentials to connect their Patient wallet.

After enrolling in the Magic Platform and opting for the free plan, the user is prompt with a dashboard, as Figure 23 shows.

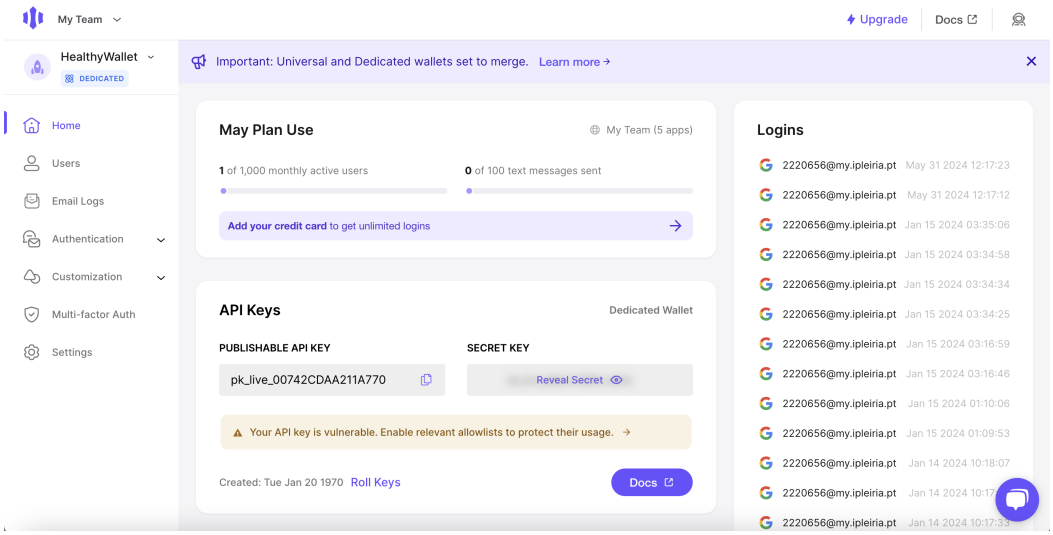


Figure 23: Magic Dashboard Overview

Magic allows up to 5 projects per free account, and they are already created by default with the same name. To prevent misuse or misconfigurations, by not having the right project selected, the project should have a dedicated name. A suitable name for this scenario would be something like “HealthyWallet”. This will represent the Patient’s Wallet in the flow of the proposed solution. Straight out-of-the-box the Publishable, or public API Key, is presented on screen. This is what is going to be used to access the *Signer* features programmatically.

Since the goal is to implement a mechanism to ease Patient Registration, a frictionless and suitable option is to use social login’s.

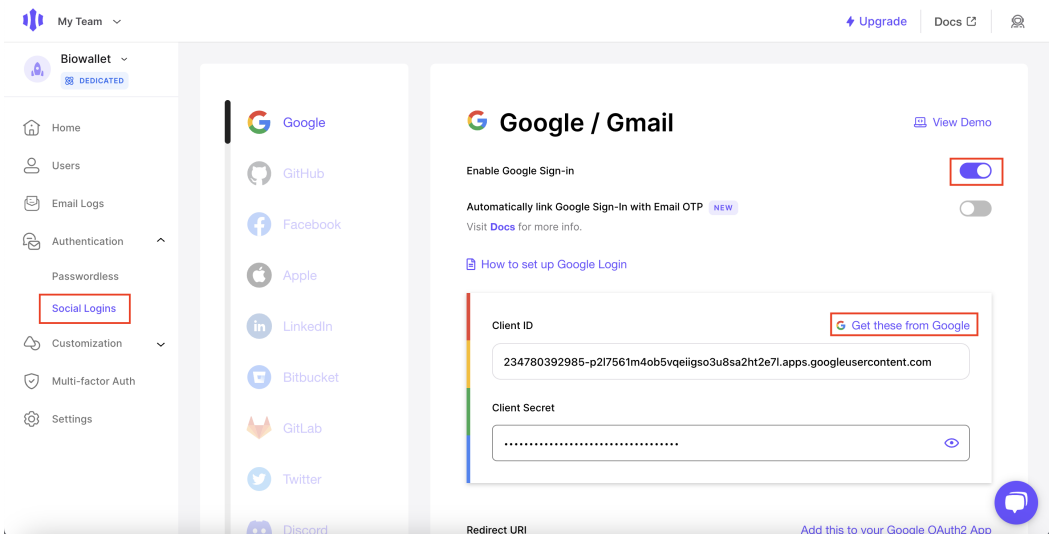


Figure 24: Magic Social Logins

As Figure 24 indicates, the process involves simply toggling the switch to "Enable Google Sign In" and proceeding to access the Google Cloud Platform or GCP to enable OAuth2.0, which enables login through Google accounts. For this, a dedicated project was created in GCP. Prior to that, the API services were enabled to create credentials to access Google's OAuth2.0 servers.

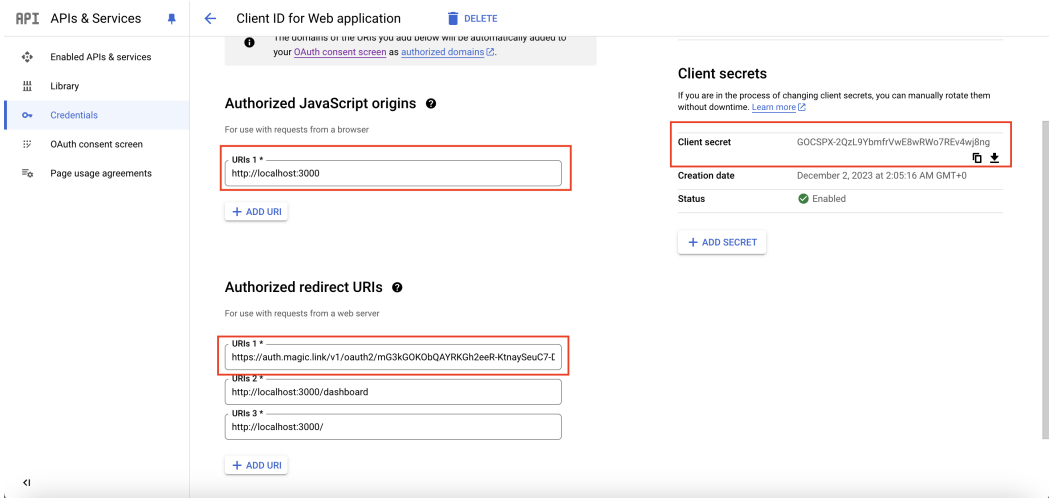


Figure 25: Configuring Google's OAuth2.0 Client

When configuring a new OAuth2.0 client, its important to define from where the requests are going to take place. In this case, and shown in Figure 25, it will the from the frontend application, that is going to be hosted and serve in `http://localhost:3000`. Another key aspect for this to work with a third-party is to allow the redirection from Magic as its shown in the "Authorized redirect URIs"

section, from Figure 25. This URI can be found in Google's Social Login page from Magic Dashboard, as seen in Figure 24. After correctly configuring the OAuth2.0 client, the `Client ID` and `Client secret` can be extracted from GCP credentials page accordingly:

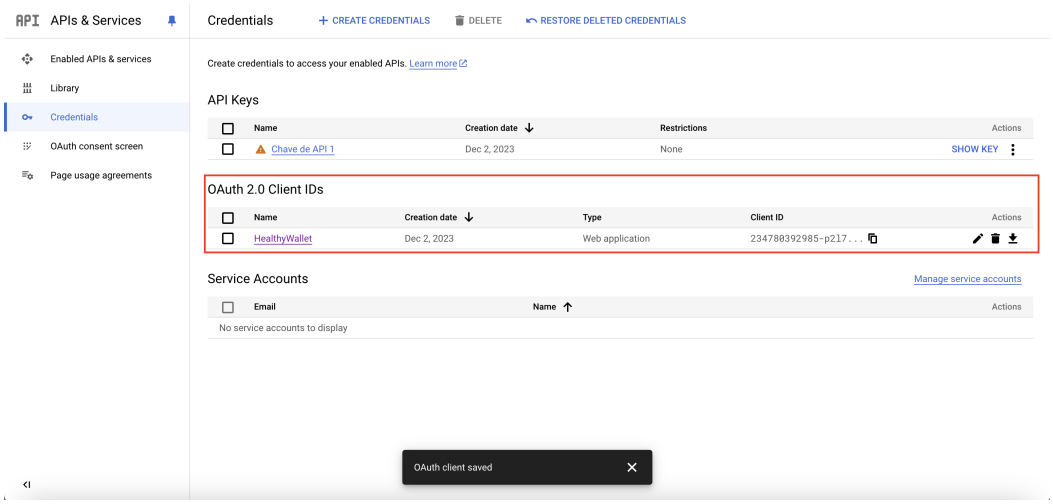


Figure 26: Google's OAuth2.0 Overview

These credentials are necessary to fill Magic's required inputs shown in Figure 26. To verify the integration between Magic and Google's OAuth2.0 client there is a section in Magic's Google Social Login Page settings for it. This can be seen in Figure 27.

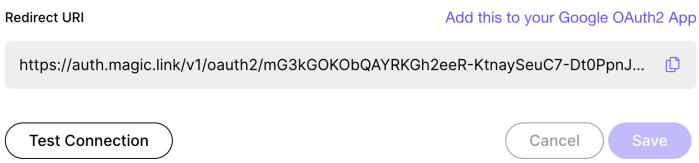


Figure 27: Testing OAuth Redirect

By testing this connection, a redirection will take place and the screen displays a success message. Finally, the only missing configuration is to use the public API Key that Magic exposes for the application created, in the `env.local` (See Appendix D).

4.18 INTEGRATION OF ALL THE SOLUTION COMPONENTS

Maintaining all these components can be challenging. Most features of both Alchemy and Magic have to work together to execute the user's requests. Since both will have its state shared across different components the best approach is to take advantage of a well know React feature called hooks[10]. In React (and Next.js), hooks are special functions that allows to use state and other features without writing a class. Hooks provide a way to handle side effects such as fetching data, setting up subscriptions, or manually changing the DOM in function components. In this context, hooks are used to encapsulate the logic for connecting to Alchemy and Magic, making it reusable across different components in the application. For that purpose, a folder named `hooks` with two files: `useAlchemyProvider.ts` and `useMagicSigner.ts` were created.

The `useAlchemyProvider` hook is designed to initialize and manage the Alchemy provider for interacting with the blockchain. It leverages the Alchemy SDK to create and manage a connection to a smart contract account, facilitating blockchain interactions. The sole purpose of the `useAlchemyProvider` hook is to provide a reusable way to connect to the Alchemy provider and manage the state of the connection, which is particularly useful in the context of building a Smart Account (ERC-4337) [71]. This hook also configures the Gas Manager API that sets the policy in the provider that was set earlier in Alchemy's dashboard.

The hook initializes an `AlchemyProvider` by specifying the chain's name (Sepolia) and the RPC URL (gathered before when initializing the app in Alchemy platform). It provides methods to connect the provider to a smart contract account using a `SmartAccountSigner` and to disconnect the provider from the account. The following Listing 10 contains the implementation of the `useAlchemyProvider` hook.

The `useAlchemyProvider` hook performs the following tasks:

- Initializes an Alchemy Provider using the given chain and `rpcUrl`.
- Provides a method to connect the provider to a smart contract account using a `SmartAccountSigner`.
- Provides a method to disconnect the provider from the Smart Contract account.

The `connect` function instantiates a Smart Account of type `LightAccount` via its constructor `LightSmartContractAccount`. The code for this is provided by

10 https://react.dev/reference/react/hooks

```
export const useAlchemyProvider = () => {
        const [provider, setProvider] = useState(new AlchemyProvider({
                chain,
                rpcUrl: getRpcUrl(),
        }));

        const connectProviderToAccount = useCallback((signer, account) => {
                const connectedProvider = provider
                .connect((provider) => new LightSmartContractAccount({
                        rpcClient: provider,
                        owner: signer,
                        chain,
                        entryPointAddress: getDefaultEntryPointAddress(chain),
                        factoryAddress: getDefaultLightAccountFactoryAddress(chain),
                        accountAddress: account,
                }))
                .withAlchemyGasManager({ policyId: gasManagerPolicyId });

                setProvider(connectedProvider);
                return connectedProvider;
        }, [provider]);

        const disconnectProviderFromAccount = useCallback(() => {
                const disconnectedProvider = provider.disconnect();
                setProvider(disconnectedProvider);
                return disconnectedProvider;
        }, [provider]);

        return { provider, connectProviderToAccount, disconnectProviderFromAccount };
};
```

Listing 10: Provider Hook

Alchemy's Account Abstraction Kit (`LightSmartContractAccount`) which accepts parameters to define the Account Owner - managed by the `Signer` (Magic), the chain where this account is going to be deployed to and the addresses for the contracts that implement the ERC-4337 norms - `EntryPoint` and `Factory` in that specific chain. The hook also maintains the state of `AlchemyProvider` and exposes functions to connect and disconnect the provider from an account. This encapsulates the logic for managing the provider, making it reusable and easy to integrate within different components of the application. Moreover this hook is memoized, which in React terms, means that this function is cached throughout the application's lifecycle, only being re-rendered upon dependency changes. These dependencies are found inside the squared brackets, e.g `[provider]`.

Overall, this means that when the provider is connected and disconnect from the user's account, it sets that new state of connect and not connected, and re-renders these functions. Furthermore, when this hook is called in any component and this one re-renders, the function that the hook defines does not get updated, unless, of course, the provider changes.

The next hook is `useMagicSigner`. It is designed to initialize and manage the *Signer*. It initializes Magic's SDK with the provided API key and OAuth extension.

```
export const useMagicSigner = () => {
        if (typeof window === "undefined") {
                return { magic: null, signer: null };
        }

        const magic = new Magic(magicApiKey, { extensions: [new OAuthExtension()], network:
        ↪  {
                        rpcUrl: "https://eth-sepolia.g.alchemy.com/v2/e**wt**c7hz\_o96KKPzV
                        ↪  j****uSGY70\_G",
                        chainId: 11155111,
        }});

        const magicClient = createWalletClient({
                transport: custom(magic.rpcProvider),
        });

        const magicSigner = new WalletClientSigner(magicClient, "magic");

        return { magic, signer: magicSigner };
};
```

Listing 11: Signer Hook

It configures the network with the Alchemy RPC URL for the Sepolia testnet, and the appropriate chain ID. It then creates a `WalletClient` using the Magic RPC provider and a `SmartAccountSigner`, using the `WalletClient`. In Listing 11 it is shown the implementation of the `useMagicSigner` hook:

The `useMagicSigner` hook firstly checks that the `window` object is available by using a guard to cancel the flow of the code execution, if the window object is `undefined`. The `window` object must be define, otherwise Web3 operations that rely on the browser won't work. Following that, it initializes the Magic SDK with the provided API key and OAuth extension to enable social logins. Also configures the network with the Alchemy RPC URL for the Sepolia testnet and the appropriate chain ID.

The next step is to instantiate a Wallet that is recognizable inside the blockchain ecosystem. For this a `magicClient` is set and creates a Web3 wallet, using the `WalletClient` type provided by viem[11] - a TypeScript interface for Ethereum blockchain by providing simple, low-level functions.

The required configuration for connecting this Wallet to the chain is by specifying whether the transport is going to be done locally (through an account address) or by JSON RPC URL. The objective of this project's case study is to use JSON RPC, thus the transport must be defined as custom and defined by the provider.

Finally, Alchemy needs to know which signer is going to be used to sign the transactions. Therefore, it allows to specify custom *Signers* that follow the Wal-

11 https://viem.sh/

letClient convention mentioned before. For that Alchemy's core package offers `WalletClientSigner` to specify just that.

Since most of the application's components interact with the Smart Wallet, therefore constantly accessing and writing to its shared state, it should be useful to implement a reusable and optimal way of managing this state. Instead of passing down the Wallet via React's props, contexts were introduced. A Context can be implemented via `useContext` hook and allows the component to subscribe and listen for changes in that specific Context. To provide a more modular and reusable solution, the implementation of a Context was implemented in a separate file, as described in Listing 12:

At first this might seem overwhelming, but as the document progresses, it starts to make more sense of how the desired state management can be achieved using React's Context. Especially because it allows for a more efficient and scalable way to share state across components without the need to pass props manually at multiple levels, ensuring cleaner and more maintainable code. The code shown in the Listing 12 creates a Context that defines what data is being shared across the children components. Most of the state data being exchanged relates to the user's personal details.

Given these specifications, the code demonstrated by Listing 12 leverages React's Context to manage the user's session, continuously monitoring for login and logout events. Whenever the session state is mutated, the shared state is also updated, resulting in all child components connected to this Context being re-rendered with the most recent user data.

The `useEffect` hook is used to ensure new information about the user is correctly fetched and used to update the children component. Since the `fetchData` method is not being called without the user directly interacting with the application, e.g. Pressing a button that calls the `fetchData` function, the application needs to know when to re-fetch data about the user.

Furthermore, `useEffect` runs the `fetchData` function once when the component mounts, because the dependency array includes `magic`, `connectProviderToAccount`, `signer`, and `provider`, which ensures the effect runs whenever any of these dependencies change.

The `fetchData` checks if Magic and its user object are initialized and whether the signer is available or not. If so, it then checks if the user is logged in, using `magic.user.isLoggedIn`. If the user is logged in, it retrieves user metadata using

```tsx
"use client";

const WalletContext = createContext<WalletContextProps>({
        provider: defaultUnset,
        login: () => Promise.resolve(),
        logout: () => Promise.resolve(),
        isLoggedIn: defaultUnset,
});

export const useWalletContext = () => useContext(WalletContext);

export const WalletContextProvider = ({ children }: { children: ReactNode }) => {
        const [ownerAddress, setOwnerAddress] = useState<Address>();
        const [scaAddress, setScaAddress] = useState<Address>();
        const [username, setUsername] = useState<string>();
        const [isLoggedIn, setIsLoggedIn] = useState<boolean>(false);

        const { magic, signer } = useMagicSigner();
        const { provider, connectProviderToAccount, disconnectProviderFromAccount }
        ↪  = useAlchemyProvider();

        const login = useCallback(async (email: string) => {
                if (!magic  !magic.user  !signer) {
                        throw new Error("Magic not initialized");
                }

                const didToken = await magic.auth.loginWithEmailOTP({ email });
                const metadata = await magic.user.getInfo();
                if (!didToken  !metadata.publicAddress  !metadata.email) {
                        throw new Error("Magic login failed");
                }

                setIsLoggedIn(true);
                connectProviderToAccount(signer);
                setUsername(metadata.email);
                setOwnerAddress(metadata.publicAddress as Address);
                setScaAddress(await provider.getAddress());
        }, [magic, connectProviderToAccount, signer, provider]);

        const logout = useCallback(async () => {
                if (!magic  !magic.user) {
                        throw new Error("Magic not initialized");
                }
                if (!(await magic.user.logout())) {
                        throw new Error("Magic logout failed");
                }
                setIsLoggedIn(false);
                disconnectProviderFromAccount();
                setUsername(undefined);
                setOwnerAddress(undefined);
                setScaAddress(undefined);
        }, [magic, disconnectProviderFromAccount]);

        useEffect(() => {
                async function fetchData() {
                        if (!magic  !magic.user  !signer) {
                                throw new Error("Magic not initialized");
                        }
                        const isLoggedIn = await magic.user.isLoggedIn();
                        if (!isLoggedIn) {
                                return;
                        }
                        const metadata = await magic.user.getInfo();
                        if (!metadata.publicAddress  !metadata.email) {
                                throw new Error("Magic login failed");
                        }
                        setIsLoggedIn(isLoggedIn);
                        connectProviderToAccount(signer);
                        setUsername(metadata.email);
                        setOwnerAddress(metadata.publicAddress as Address);
                        setScaAddress(await provider.getAddress());
                }
                fetchData();
        }, [magic, connectProviderToAccount, signer, provider]);

        (...) // See Appendix C for a detailed explanation of the render function.
};
```

Listing 12: Wallet Context Provider

```
export default function Home() {
        return (
                <WalletContextProvider>
                <Root />
                </WalletContextProvider>
        );
}
```

Listing 13: Wallet Context Wrapper

`magic.user.getInfo`. Finally, it updates the Context state with the user's email, public address, and smart contract account address.

To clarify the application's described flow, Figure 28 illustrates how React's Context mechanism is used to share and update state across components.

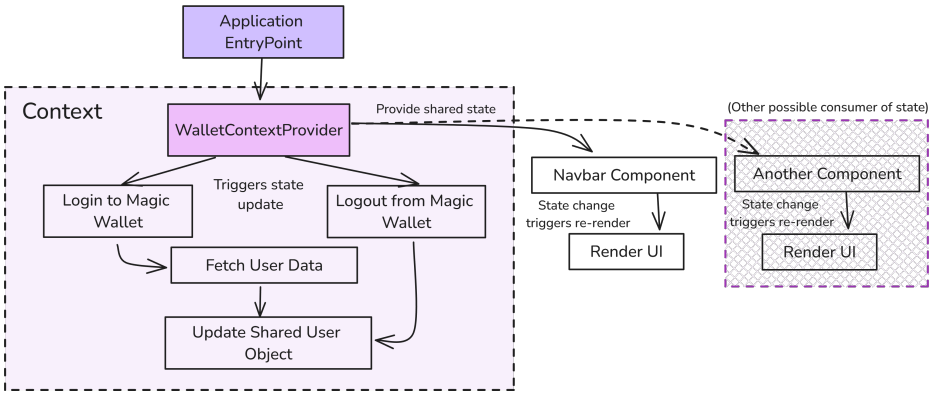


Figure 28: State Management using React Context

The following step is to wrap the application with the newly created Context. For that purpose, the so called application *Entry point* is going to import `WalletContext` and wrap itself with it (see Listing 13). Since NextJS is configured to use routing based on directories (see Figure 17) the entry point is the `page.tsx` located in the `app` directory.

The `Root` component is imported from the `@/components/Root` module. This component represents the main content of the application's interface. This code exports the default function called `Home()` which defines the initial component for the application. This is simply the function that is called whenever the user navigates to the \ root URL. The application returns a `View` based on that exported function.

The next step is to define a component where the user can check his login status and manage it. With that purpose in mind, the `NavBar` component was created under the `\components` folder:

```tsx
export default function Navbar() {
        const [isLoggingIn, setIsLoggingIn] = useState<boolean>(false);
        const [isLoggingOut, setIsLoggingOut] = useState<boolean>(false);
        const [email, setEmail] = useState<string>("");
        const { magic, signer } = useMagicSigner();
        const { provider } = useAlchemyProvider();

        const handleSocialLogin = async () => {
                const baseURL = new URL("/", window.location.origin).href;
                try {
                        await (magic as any).oauth.loginWithRedirect({
                                provider: "google",
                                redirectURI: baseURL,
                        });
                } catch (err) {
                        console.error(err);
                }
        };

        const finishSocialLogin = async () => {
                try {
                        if (!magic  !magic.user  !signer) {
                                throw new Error("Magic not initialized");
                        }

                        const result = await magic.oauth.getRedirectResult();
                        const owner = await provider?.account?.getAddress();
                        console.log(result);
                        console.log(owner);
                } catch (err) {
                        console.error(err);
                }
        };

        useEffect(() => {
                finishSocialLogin();
        }, []); // eslint-disable-line react-hooks/exhaustive-deps

        const { isLoggedIn, login, logout, username, scaAddress } = useWalletContext();

        const openModal = useCallback(() => {
                setIsLoggingIn(true);
        }, []);

        const closeModal = useCallback(() => {
                setIsLoggingIn(false);
        }, []);

        const onEmailChange = useCallback((e: React.ChangeEvent<HTMLInputElement>) => {
                e.preventDefault();
                setEmail(e.target.value);
        }, []);

        const handleLogin = useCallback(async () => {
                await login(email);
                setIsLoggingIn(false);
                setEmail("");
        }, [login, email]);

        const handleLogout = useCallback(async () => {
                setIsLoggingOut(true);
                await logout();
                setIsLoggingOut(false);
        }, [logout]);

        (...) // See Appendix B for a detailed explanation of the render function.
}
```

Listing 14: Navbar Component

The component referenced in Listing 14 uses the `useState` hook to manage state variables for the login process (`isLoggingIn`), logout process (`isLoggingOut`), and user email (`email`). These states help track whether the login/logout processes are active and store the user's email input.

The `useMagicSigner` and `useAlchemyProvider` hooks initialize the `magic`, `signer`, and `provider` objects from Magic and Alchemy, respectively. These hooks facilitate interaction with Magic for authentication and Alchemy for blockchain interactions.

The `handleSocialLogin` function initiates the social login process using Magic's OAuth with Google. It constructs the base URL for redirection and calls Magic's `loginWithRedirect` method.

The `finishSocialLogin` function retrieves the result of the OAuth2.0 login process. It checks if Magic and the signer are properly initialized, then fetches the OAuth2.0 result and the owner's address from the provider. This function is called within a `useEffect` hook to ensure it runs when the component mounts, completing the login process and fetching user information.

The `useEffect` hook invokes `finishSocialLogin` when the component mounts. This ensures that the login process is completed and user information is fetched.

The `openModal` and `closeModal` functions manage the state of the login modal. These functions toggle the `isLoggingIn` state to show or hide the modal.

The `onEmailChange` function updates the `email` state with the user's input. This function ensures that the email input field correctly reflects the user's input.

The `handleLogin` function triggers the login process using the provided email. It calls the `login` function, from the Wallet Context, and updates the state variables accordingly.

The `handleLogout` function handles the logout process, setting the `isLoggingOut` state to true and invoking the `logout` function from the Wallet Context.

To sum, the `NavBar` component returns a `JSX` structure, which can be seen further in Appendix B, that includes the application title, login and logout buttons, and a login modal. The login button initiates the social login process via Google's OAuth2.0 client, and the logout button allows the user to end his session. Lastly, conditional rendering is used to display different content based on the user's authentication status.

Incorporating all these components, the final structure and functionality of the application can be observed in Figure 29

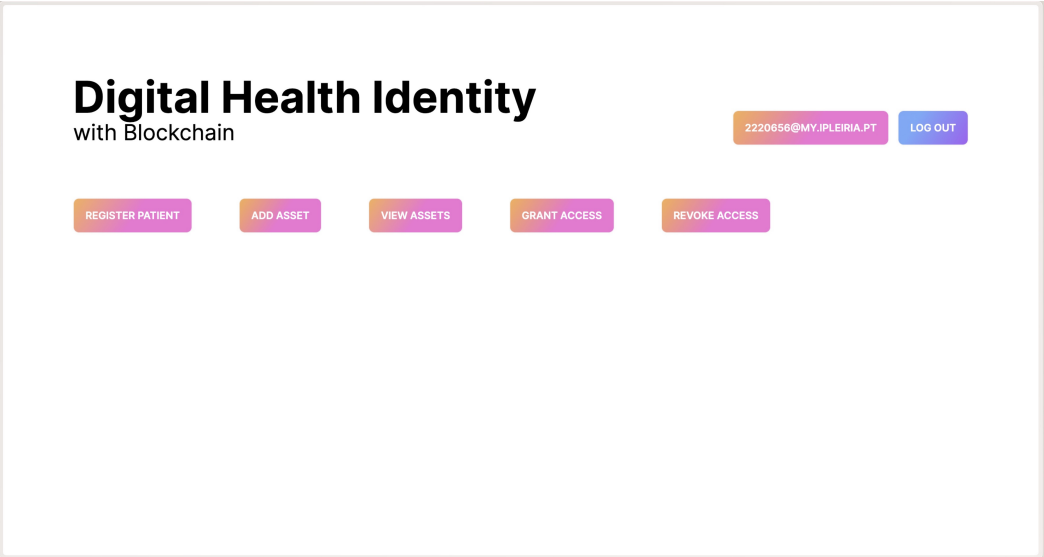


Figure 29: Final Application's aspect

5

TESTING AND VALIDATION

Like in any other code platform, ensuring the correctness and security of the code is crucial. Smart contracts are immutable once deployed, which means any bugs or vulnerabilities can have irreversible consequences. To mitigate such risks, it is essential to perform thorough testing before deploying the contract to a live blockchain network.

An effective methodology for validating the functionality and reliability of smart contracts is unit testing [88]. Unit testing involves writing tests that focus on individual components or functions of the smart contract, isolating them from other parts of the system. By doing so, the solution can be verified and assert that each part of the contract behaves as expected under various conditions.

This section presents a series of unit tests designed to validate the functionality of the `AssetManagement` Smart Contract. These tests ensure that the contract correctly manages medical asset IDs, enforces access control, and handles the granting and revoking of permissions securely.

The test suite uses the `Hardhat`[1] testing framework along with `Chai`[2] for assertions. It begins by setting up the necessary environment before each test using the `beforeEach` hook. The contract is deployed and three Ethereum accounts, `patient`, `healthcareProvider`, and `other`, are initialized. This ensures that each test starts with a fresh deployment of the contract and isolated account states.

```
1  beforeEach(async function () {
2      const AssetManagement = await
       ↪  ethers.getContractFactory("AssetManagement");
3      assetManagement = await AssetManagement.deploy();
4      await assetManagement.waitForDeployment();
5
6      [patient, healthcareProvider, other] = await ethers.getSigners();
7      await assetManagement.connect(patient).registerPatient();
8  });
```

Listing 15: Pre-Test configuration

The `beforeEach` function listed in Listing 15, first obtains the contract factory using `ethers.getContractFactory`, and then deploys the contract with `deploy()`.

1 https://hardhat.org/docs
2 https://www.chaijs.com/

The `waitForDeployment()` function ensures that the deployment completes before the test proceeds. The accounts are initialized using `getSigners()`, and the `registerPatient()` function is called to register the patient in the contract.

A utility function `generateHash` is also defined, and listed in Listing 16, which generates a SHA-256 hash of a given asset ID. This function simulates the process of securely hashing asset identifiers before they are stored on the blockchain.

```
function generateHash(realId) {
        return CryptoJS.SHA256(realId).toString();
}
```

Listing 16: Generate Hash

The first test case, *"Should store and verify hashed asset IDs"* listed in Listing 17, checks whether the contract can correctly store and retrieve a hashed asset ID. The asset ID is hashed using the `generateHash` function, stored using the `addMedicalAssetId` function, and then retrieved to verify that the hashed ID was stored correctly.

```
it("Should store and verify hashed asset IDs", async function () {
        const realId = "asset-123";
        const hashedId = generateHash(realId);

        await assetManagement
        .connect(patient)
        .addMedicalAssetId(hashedId);

        const allAssetIds = await assetManagement
        .connect(patient)
        .viewAllAssetIds();
        expect(allAssetIds)
        .to
        .include(hashedId);

        console.log("Stored hashed ID: ", hashedId);
});
```

Listing 17: Should store and verify hashed asset IDs

The next test, *"Should allow patient to grant access to a specific asset ID to a healthcare provider"* listed in Listing 18, verifies that a patient can grant access to a healthcare provider, for a specific asset ID. The patient first stores an asset ID, then grants access to the healthcare provider using the `grantAccessToAssetId` function. The test then checks if the healthcare provider can access the asset ID by calling the `viewMedicalAssetId` function.

```
it("Should allow patient to grant access to a specific asset ID to a healthcare provider", async function () {
        const assetId = "asset1";
        await assetManagement
        .connect(patient)
        .addMedicalAssetId(assetId);
        await assetManagement
        .connect(patient)
        .grantAccessToAssetId(assetId, healthcareProvider.address);

        const retrievedAssetId = await assetManagement.connect(healthcareProvider)
        .viewMedicalAssetId(patient.address, assetId);
        expect(retrievedAssetId).to.equal(assetId);
});
```

Listing 18: Should allow patient to grant access to a specific asset ID to a healthcare provider

The third test, *"Should prevent unauthorized entities from viewing a specific medical asset ID"*, listed in Listing 19, ensures that unauthorized entities cannot access a specific asset ID. In this test, the patient stores an asset ID, and the test attempts to access it using an unauthorized account (`other`). The test expects the contract to revert with an error message indicating that the entity is not authorized to access the asset ID.

```
it("Should prevent unauthorized entities from viewing a specific medical asset ID", async function () {
        const assetId = "asset2";
        await assetManagement
        .connect(patient)
        .addMedicalAssetId(assetId);

        await expect(
        assetManagement
        .connect(other)
        .viewMedicalAssetId(patient.address, assetId)
        ).to.be.revertedWith("Not authorized for this asset ID");
});
```

Listing 19: Should prevent unauthorized entities from viewing a specific medical asset ID

The test case, *"Should allow patient to revoke access to a specific asset ID from a healthcare provider"* listed in Listing 20, tests the contract's ability to revoke access to an asset ID that was previously granted. After revoking access, the test ensures that the healthcare provider can no longer access the asset ID. The test uses the `revokeAccessToAssetId` function and verifies access using `viewMedicalAssetId`.

```
it("Should allow patient to revoke access to a specific asset ID from a
↪  healthcare provider", async function () {
        const assetId = "asset3";
        await assetManagement
        .connect(patient)
        .addMedicalAssetId(assetId);
        await assetManagement
        .connect(patient)
        .grantAccessToAssetId(assetId, healthcareProvider.address);
        await assetManagement
        .connect(patient)
        .revokeAccessToAssetId(assetId, healthcareProvider.address);

        await expect(
        assetManagement
        .connect(healthcareProvider)
        .viewMedicalAssetId(patient.address, assetId)
        ).to.be.revertedWith("Not authorized for this asset ID");
});
```

Listing 20: Should allow patient to revoke access to a specific asset ID from a healthcare provider

Finally, the test case *"Should allow patient to view all their medical asset IDs"* listed in Listing 21 verifies that a patient can retrieve a list of all their stored asset IDs. The patient stores multiple asset IDs, and the test checks if they can be retrieved correctly using the `viewAllAssetIds` function.

```
it("Should allow patient to view all their medical asset IDs", async
↪  function () {
        const assetId1 = "asset1";
        const assetId2 = "asset2";
        await assetManagement
        .connect(patient)
        .addMedicalAssetId(assetId1);
        await assetManagement
        .connect(patient)
        .addMedicalAssetId(assetId2);

        const assetIds = await assetManagement
        .connect(patient).viewAllAssetIds();
        expect(assetIds).to.include(assetId1);
        expect(assetIds).to.include(assetId2);
        console.log("All stored asset IDs: ", assetIds);
});
```

Listing 21: Should allow patient to view all their medical asset IDs

6

CONCLUSIONS

The research conducted in this thesis explores the application of blockchain technologies, specifically Account Abstraction and Zero-Knowledge Proofs (ZKP), to address critical challenges in the healthcare sector, namely identity management and access control for medical data. The proposed architectural solution not only resolves these challenges but also enhances privacy and user experience in a decentralized healthcare ecosystem. By leveraging blockchain technology, this work ensures that patients have full control over their data, allowing them to selectively grant or revoke access to healthcare providers without exposing sensitive relationships on a public ledger. This approach effectively mitigates the risk of data mining and unauthorized data aggregation, which are prevalent concerns when patient data is stored in decentralized systems.

The architecture developed within this research enables patients to securely engage with healthcare providers in a private and controlled manner. Using ZKP, the system validates patient-provider relationships and permissions without disclosing any identifiable data, thus preserving the confidentiality of interactions. This patient-centric model empowers individuals to manage their data more effectively, reinforcing the concept of patients as the primary owners of their health information. Additionally, the proposed solution supports the dynamic management of access rights, including time-limited authorizations and the ability for patients to restrict shared information according to their preferences, further strengthening the privacy framework.

Throughout the research, extensive efforts were made to compare various blockchain solutions, including public, private, and consortium blockchains, to determine the most suitable approach for healthcare applications. The study highlighted the limitations of existing implementations, such as EBSI and Hedera, in terms of privacy, scalability, and interoperability. The choice of Ethereum as the underlying blockchain was driven by its open nature, community support, and extensive tooling, which collectively provide a robust foundation for implementing secure and scalable healthcare solutions. The integration of advanced cryptographic techniques, such as

ZKP, was pivotal in addressing the specific needs of identity and access management while avoiding the pitfalls associated with traditional blockchain-based systems.

One of the notable achievements of this research is the development of a system that not only secures patient data but also paves the way for innovative data-sharing models. Future research could build upon this foundation to explore the integration of healthcare data protocols, such as FHIR, into blockchain systems, creating seamless connections between decentralized platforms and existing healthcare infrastructures. Furthermore, the scalability of the solution in terms of financial and computational costs remains an area of ongoing exploration, particularly concerning the deployment of ZKP and the storage requirements for medical records on the blockchain.

Looking ahead, an interesting avenue for future work involves the development of mechanisms that allow patients to grant access to their data for clinical research purposes under explicit agreements. This could include the implementation of smart contracts that facilitate economic compensation for patients who choose to share their data, thereby creating a mutually beneficial arrangement for all stakeholders within the healthcare ecosystem. Such advancements would not only enhance patient autonomy but also contribute to a more sustainable and transparent model for medical research and innovation.

In conclusion, this thesis demonstrates that the thoughtful integration of blockchain technology and advanced cryptographic methods can significantly improve the management of identity and access in healthcare systems. By prioritizing patient privacy and control, the proposed solution lays the groundwork for a new era of secure and decentralized healthcare platforms that can meet the complex needs of modern medical data management. The insights gained from this research not only offer valuable contributions to the broader discourse on blockchain applications in healthcare, underscoring the potential of decentralized technologies to transform the management and sharing of sensitive information in a privacy-preserving manner, but also in the personal growth of the author. All of these innovative concepts and technologies further enrich the authors' already extensive expertise, primarily cultivated through their academic endeavors.

This work has also resulted in the production of the following academic contributions:

- A conference paper titled *"Architecture for Health Private Data Sharing using Blockchain"*, which has been accepted for presentation at the HCIST conference;

- A journal article, titled *"Using Smart Contracts and Zero-Knowledge Proofs for Privacy-Preserving Medical Data Sharing Agreements"*, submitted and currently under review for publication in the journal Blockchain: Research and Applications.

BIBLIOGRAPHY

[1] S. Goldwasser, S. Micali, and C. Rackoff, "Knowledge complexity of interactive proof-systems.", *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 291–304, 1985, ISSN: 07349025. DOI: 10.1145/22145.22178. [Online]. Available: https://dl.acm.org/doi/10.1145/22145.22178.

[2] M. E. Rana, M. Jayabalan, and M. E. R. M. J. M. A. Aasif, "Privacy preserving anonymization techniques for patient data: An overview", 2016. [Online]. Available: https://www.researchgate.net/publication/318720516.

[3] S. Rana, F. K. Parast, B. Kelly, Y. Wang, and K. B. Kent, "A comprehensive survey of cryptography key management systems", *Journal of Information Security and Applications*, vol. 78, p. 103 607, Nov. 2023, ISSN: 2214-2126. DOI: 10.1016/J.JISA.2023.103607.

[4] C. Jacob, A. Y. Abdin, and H. Taherdoost, "Privacy and security of blockchain in healthcare: Applications, challenges, and future perspectives", *Sci 2023, Vol. 5, Page 41*, vol. 5, p. 41, 4 Oct. 2023, ISSN: 2413-4155. DOI: 10.3390/SCI5040041. [Online]. Available: https://www.mdpi.com/2413-4155/5/4/41.

[5] E. R. Villarreal, J. Garcia-Alonso, E. Moguel, and J. A. H. Alegria, "Blockchain for healthcare management systems: A survey on interoperability and security", *IEEE Access*, vol. 11, pp. 5629–5652, 2023, ISSN: 21693536. DOI: 10.1109/ACCESS.2023.3236505.

[6] G. A. Oliva, A. E. Hassan, and Z. M. (Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform", *Empirical Software Engineering*, vol. 25, pp. 1864–1904, 3 May 2020, ISSN: 15737616. DOI: 10.1007/S10664-019-09796-5. [Online]. Available: https://link.springer.com/article/10.1007/s10664-019-09796-5.

[7] *Erc-4337: Account abstraction using alt mempool*. Accessed: Nov. 19, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-4337.

[8] A. K. Singh, I. U. Hassan, G. Kaur, S. Kumar, and Anmol, "Account abstraction via singleton entrypoint contract and verifying paymaster", *Proceedings of the 2nd International Conference on Edge Computing and Applications,*

ICECAA 2023, pp. 1598–1605, 2023. DOI: 10.1109/ICECAA58104.2023.10212316.

[9] T. Heart, O. Ben-Assuli, and I. Shabtai, "A review of phr, emr and ehr integration: A more personalized healthcare and public health policy", *Health Policy and Technology*, vol. 6, pp. 20–25, 1 Mar. 2017, ISSN: 2211-8837. DOI: 10.1016/J.HLPT.2016.08.002.

[10] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", [Online]. Available: www.bitcoin.org.

[11] D. Burkhardt, M. Werling, and H. Lasi, "Distributed ledger", *2018 IEEE International Conference on Engineering, Technology and Innovation, ICE/ITMC 2018 - Proceedings*, Aug. 2018. DOI: 10.1109/ICE.2018.8436299. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8436299.

[12] H. Saeed *et al.*, "Blockchain technology in healthcare: A systematic review", *PLoS ONE*, vol. 17, 4 Apr. 2022, ISSN: 19326203. DOI: 10.1371/JOURNAL.PONE.0266462. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9000089/.

[13] A. S. Rajasekaran, M. Azees, and F. Al-Turjman, "A comprehensive survey on blockchain technology", *Sustainable Energy Technologies and Assessments*, vol. 52, p. 102 039, Aug. 2022, ISSN: 2213-1388. DOI: 10.1016/J.SETA.2022.102039.

[14] B. Lashkari and P. Musilek, "A comprehensive review of blockchain consensus mechanisms", *IEEE Access*, vol. 9, pp. 43 620–43 652, 2021, ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3065880.

[15] S. ; Zhou *et al.*, "A systematic review of consensus mechanisms in blockchain", *Mathematics 2023, Vol. 11, Page 2248*, vol. 11, p. 2248, 10 May 2023, ISSN: 2227-7390. DOI: 10.3390/MATH11102248. [Online]. Available: https://www.mdpi.com/2227-7390/11/10/2248/.

[16] A. Zhang and X. Lin, "Towards secure and privacy-preserving data sharing in e-health systems via consortium blockchain", *Journal of Medical Systems*, vol. 42, pp. 1–18, 8 Aug. 2018, ISSN: 1573689X. DOI: 10.1007/S10916-018-0995-5. [Online]. Available: https://link.springer.com/article/10.1007/s10916-018-0995-5.

[17] H. Zhu, Y. Guo, and L. Zhang, "An improved convolution merkle tree-based blockchain electronic medical record secure storage scheme", *Journal of Information Security and Applications*, vol. 61, p. 102 952, Sep. 2021, ISSN: 2214-2126. DOI: 10.1016/J.JISA.2021.102952.

[18] R. C. Merkle, "A digital signature based on a conventional encryption function", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 293 LNCS, pp. 369–378, 1988, ISSN: 1611-3349. DOI: 10.1007/3-540-48184-2_32. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-48184-2_32.

[19] "A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges", *Decision Analytics Journal*, vol. 9, p. 100 344, Dec. 2023, ISSN: 2772-6622. DOI: 10.1016/J.DAJOUR.2023.100344.

[20] *Ethereum whitepaper | ethereum.org*. Accessed: Oct. 11, 2023. [Online]. Available: https://ethereum.org/whitepaper.

[21] A. I. Sanka and R. C. Cheung, "A systematic review of blockchain scalability: Issues, solutions, analysis and future research", *Journal of Network and Computer Applications*, vol. 195, p. 103 232, Dec. 2021, ISSN: 1084-8045. DOI: 10.1016/J.JNCA.2021.103232.

[22] W. J. Buchanan *et al.*, "The future of integrated digital governance in the eu: Ebsi and glass", Dec. 2022. [Online]. Available: https://arxiv.org/abs/2212.03218v2.

[23] *White papers – hyperledger foundation*. Accessed: Oct. 9, 2023. [Online]. Available: https://www.hyperledger.org/learn/white-papers.

[24] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains", [Online]. Available: www.hyperledger.org.

[25] H. W. Marar and R. W. Marar, "Hybrid blockchain", *Jordanian Journal of Computers and Information Technology*, vol. 6, pp. 317–325, 4 2020, ISSN: 24151076. DOI: 10.5455/JJCIT.71-1589089941.

[26] A. Arooj, M. S. Farooq, and T. Umer, "Unfolding the blockchain era: Timeline, evolution, types and real-world applications", *Journal of Network and Computer Applications*, vol. 207, p. 103 511, Nov. 2022, ISSN: 1084-8045. DOI: 10.1016/J.JNCA.2022.103511.

[27] P. K. Paul, P. S. Aithal, R. Saavedra, and S. Ghosh, "Blockchain technology and its types-a short review", 2021. Accessed: May 16, 2024. [Online]. Available: https://ssrn.com/abstract=4050933.

[28] O. Dib, A. Durand, K.-L. Brousmiche, E. Thea, and B. Hamida, "Consortium blockchains: Overview, applications and challenges", *International Journal on Advances in Telecommunications*, 2 2018. [Online]. Available: https://www.researchgate.net/publication/328887130_Consortium_Blockchains_Overview_Applications_and_Challenges.

[29] K. Lei, Q. Zhang, L. Xu, and Z. Qi, "Reputation-based byzantine fault-tolerance for consortium blockchain", *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2018-December, pp. 604–611, Jul. 2018, ISSN: 15219097. DOI: 10.1109/PADSW.2018.8644933.

[30] M. Xu, X. Chen, and G. Kou, "A systematic review of blockchain", *Financial Innovation*, vol. 5, pp. 1–14, 1 Dec. 2019, ISSN: 21994730. DOI: 10.1186/S40854-019-0147-Z. [Online]. Available: https://link.springer.com/article/10.1186/s40854-019-0147-z.

[31] M. N. Alenezi, H. Alabdulrazzaq, and N. Q. Mohammad, "Symmetric encryption algorithms: Review and evaluation study", *Article in International Journal of Communication Networks and Information Security*, vol. 12, 2 2020. [Online]. Available: https://www.researchgate.net/publication/349324592.

[32] *Erc-6900: Modular smart contract accounts and plugins*. Accessed: Nov. 18, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-6900.

[33] E. Onica and M. Georgic, "Can smart contracts become smart?: An overview of transaction impact on ethereum dapp engineering", *DICG 2023 - Proceedings of the 2023 4th International Workshop on Distributed Infrastructure for Common Good, Part of: MIDDLEWARE 2023*, pp. 31–36, Dec. 2023. DOI: 10.1145/3631310.3633492. [Online]. Available: https://dl.acm.org/doi/10.1145/3631310.3633492.

[34] A. Antonopoulos and G. W. Ph.D, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2019, ISBN: 9781491971949. [Online]. Available: https://github.com/ethereumbook/ethereumbook/blob/develop/book.asciidoc.

[35] S. Tikhomirov, "Ethereum: State of knowledge and research perspectives", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10723 LNCS, pp. 206–221, 2018, ISSN: 1611-3349. DOI: 10.1007/978-3-319-75650-9_14.

[Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-75650-9_14.

[36] A. Petcu, B. Pahontu, M. Frunzete, and D. A. Stoichescu, "A secure and decentralized authentication mechanism based on web 3.0 and ethereum blockchain technology", *Applied Sciences 2023, Vol. 13, Page 2231*, vol. 13, p. 2231, 4 Feb. 2023, ISSN: 2076-3417. DOI: 10.3390/APP13042231. [Online]. Available: https://www.mdpi.com/2076-3417/13/4/2231/.

[37] J. H. Khor, M. Sidorov, and S. A. B. Zulqarnain, "Scalable lightweight protocol for interoperable public blockchain-based supply chain ownership management", *Sensors 2023, Vol. 23, Page 3433*, vol. 23, p. 3433, 7 Mar. 2023, ISSN: 1424-8220. DOI: 10.3390/S23073433. [Online]. Available: https://www.mdpi.com/1424-8220/23/7/3433/.

[38] *Introduction to zk-snarks | consensys*. Accessed: Nov. 17, 2023. [Online]. Available: https://consensys.io/blog/introduction-to-zk-snarks.

[39] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, "A review of zk-snarks", Feb. 2022. [Online]. Available: https://arxiv.org/abs/2202.06877v4.

[40] *Iso/tr20514:2005(en), health informatics— electronic health record— definition, scope and context*. Accessed: May 28, 2024. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:tr:20514:ed-1:v1:en.

[41] *Health level seven international - homepage | hl7 international*. Accessed: May 28, 2024. [Online]. Available: https://www.hl7.org/.

[42] *Hl7 standards product brief - fhir® (hl7 fast healthcare interoperability resources) | hl7 international*. Accessed: May 28, 2024. [Online]. Available: https://www.hl7.org/implement/standards/product_brief.cfm?product_id=491.

[43] *Openehr - specifications start page*. Accessed: May 28, 2024. [Online]. Available: https://specifications.openehr.org/start.

[44] *Openehr vs iso 13606. healthcare standards and specifications... | by alastair allen | medium*. Accessed: May 28, 2024. [Online]. Available: https://medium.com/@alastairallen/openehr-vs-iso13606-10cbf0690555.

[45] L. Min, Q. Tian, X. Lu, and H. Duan, "Modeling ehr with the openehr approach: An exploratory study in china philip payne", *BMC Medical Informatics and Decision Making*, vol. 18, pp. 1–15, 1 Aug. 2018, ISSN: 14726947. DOI: 10.1186/S12911-018-0650-6. [Online]. Available: https://link.springer.com/articles/10.1186/s12911-018-0650-6.

[46] I. of Medicine (US) Committee on Data Standards for Patient Safety, P. Aspden, J. M. Corrigan, J. Wolcott, and S. M. Erickson, "Health care data standards", 2004. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK216088/.

[47] *Home | snomed international*. Accessed: Apr. 23, 2024. [Online]. Available: https://www.snomed.org/.

[48] *Home – loinc*. Accessed: Apr. 23, 2024. [Online]. Available: https://loinc.org/.

[49] *Hipaa home | hhs.gov*. Accessed: Apr. 23, 2024. [Online]. Available: https://www.hhs.gov/hipaa/index.l.

[50] *What is ebsi - ebsi -*. Accessed: Oct. 8, 2023. [Online]. Available: https://ec.europa.eu/digital-building-blocks/sites/display/EBSI/What+is+EBSI.

[51] *Ibft 2.0 | besu documentation*. Accessed: Oct. 7, 2024. [Online]. Available: https://besu.hyperledger.org/private-networks/how-to/configure/consensus/ibft.

[52] E. Tan, E. Lerouge, J. D. Caju, and D. D. Seuil, "Verification of education credentials on european blockchain services infrastructure (ebsi): Action research in a cross-border use case between belgium and italy", *Big Data and Cognitive Computing*, vol. 7, p. 79, 2 Jun. 2023, ISSN: 25042289. DOI: 10.3390/BDCC7020079. [Online]. Available: https://www.mdpi.com/2504-2289/7/2/79/.

[53] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management", *Proceedings - 2016 2nd International Conference on Open and Big Data, OBD 2016*, pp. 25–30, Sep. 2016. DOI: 10.1109/OBD.2016.11.

[54] *What is drm? digital rights management explained | fortinet*. Accessed: Apr. 20, 2024. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/digital-rights-management-drm.

[55] B. Shen, J. Guo, and Y. Yang, "Medchain: Efficient healthcare data sharing via blockchain", *Applied Sciences 2019, Vol. 9, Page 1207*, vol. 9, p. 1207, 6 Mar. 2019, ISSN: 2076-3417. DOI: 10.3390/APP9061207. [Online]. Available: https://www.mdpi.com/2076-3417/9/6/1207/.

[56] P. Zhang, J. White, D. C. Schmidt, G. Lenz, and S. T. Rosenbloom, "Fhirchain: Applying blockchain to securely and scalably share clinical data", *Computational and Structural Biotechnology Journal*, vol. 16, pp. 267–278, Jan. 2018, ISSN: 2001-0370. DOI: 10.1016/J.CSBJ.2018.07.004.

[57] *Interoperability roadmap | healthit.gov*. Accessed: Apr. 21, 2024. [Online]. Available: https://www.healthit.gov/topic/interoperability/interoperability-roadmap.

[58] *The c4 model for visualising software architecture*. Accessed: Apr. 21, 2024. [Online]. Available: https://c4model.com/.

[59] *Home - truffle suite*. Accessed: Oct. 29, 2023. [Online]. Available: https://archive.trufflesuite.com/.

[60] *Hardhat | ethereum development environment for professionals by nomic foundation*. Accessed: Oct. 29, 2023. [Online]. Available: https://hardhat.org/.

[61] *Remix - ethereum ide*. Accessed: Oct. 29, 2023. [Online]. Available: https://remix.ethereum.org/.

[62] *Ethereum virtual machine (evm) | ethereum.org*. Accessed: Oct. 29, 2023. [Online]. Available: https://ethereum.org/en/developers/docs/evm/.

[63] *What are testnets?* Accessed: Nov. 2, 2023. [Online]. Available: https://www.alchemy.com/overviews/what-are-testnets.

[64] *Sepolia resources – information hub for the sepolia testnet*. Accessed: Nov. 10, 2023. [Online]. Available: https://sepolia.dev/.

[65] *Choosing a web3 network*. Accessed: Nov. 10, 2023. [Online]. Available: https://docs.alchemy.com/docs/choosing-a-web3-network#which-ethereum-testnet-should-i-use.

[66] *Alchemy sdk quickstart*. Accessed: Nov. 8, 2023. [Online]. Available: https://docs.alchemy.com/reference/alchemy-sdk-quickstart.

[67] *Magic home | magic*. Accessed: Nov. 13, 2023. [Online]. Available: https://magic.link/docs/home/welcome.

[68] S. Rahman, N. Hossain, M. A. Hossain, M. Hossain, and M. Sohag, "Oauth 2.0: A framework to secure the oauth-based service for packaged web application", pp. 92–139, Jan. 2020. DOI: 10.4018/978-1-7998-3355-0.ch005.

[69] *Auth0 | why choose social login?* Accessed: Nov. 12, 2023. [Online]. Available: https://auth0.com/resources/whitepapers/why-choose-social-login.

[70] zokrates, *Introduction - zokrates*. Accessed: Dec. 17, 2023. [Online]. Available: https://zokrates.github.io/.

[71] *Erc 4337*. Accessed: Nov. 2, 2023. [Online]. Available: https://www.erc4337.io/.

[72] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, "A survey on zero-knowledge proof in blockchain", *IEEE Network*, vol. 35, pp. 198–205, 4 Jul. 2021, ISSN: 1558156X. DOI: 10.1109/MNET.011.2000473.

[73] *What is a zero-knowledge proof?* Accessed: Nov. 18, 2024. [Online]. Available: https://chain.link/education/zero-knowledge-proof-zkp#toc-first.

[74] D. Capko, S. Vukmirovic, and N. Nedic, "State of the art of zero-knowledge proofs in blockchain", *2022 30th Telecommunications Forum, TELFOR 2022 - Proceedings*, 2022. DOI: 10.1109/TELFOR56187.2022.9983760.

[75] *Ganache | overview - truffle suite*. Accessed: Nov. 11, 2023. [Online]. Available: https://archive.trufflesuite.com/docs/ganache/.

[76] *Introduction - foundry book*. Accessed: Nov. 11, 2023. [Online]. Available: https://book.getfoundry.sh/.

[77] *Ape framework | apeworx, ltd.* Accessed: Nov. 11, 2023. [Online]. Available: https://www.apeworx.io/framework/.

[78] *Typescript: Javascript with syntax for types.* Accessed: Nov. 10, 2023. [Online]. Available: https://www.typescriptlang.org/.

[79] *Home | metamask developer documentation*. Accessed: Dec. 1, 2023. [Online]. Available: https://docs.metamask.io/.

[80] *Ethereum (eth) blockchain explorer*. Accessed: Dec. 15, 2023. [Online]. Available: https://etherscan.io/.

[81] J. Voas and N. Kshetri, "Lost and never found", *Computer*, vol. 54, pp. 12–13, 07 Jul. 2021, ISSN: 0018-9162. DOI: 10.1109/MC.2021.3055712. [Online]. Available: https://ieeexplore.ieee.org/document/9473212.

[82] N. Koblitz, A. Menezes, and S. Vanstone, "The state of elliptic curve cryptography", *Designs, Codes, and Cryptography*, vol. 19, pp. 173–193, 2-3 2000, ISSN: 09251022. DOI: 10.1023/A:1008354106356. [Online]. Available: https://www.researchgate.net/publication/220638302_The_State_of_Elliptic_Curve_Cryptography.

[83] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems",

[84] J. Liu and M. Manulis, "Fast snark-based non-interactive distributed verifiable random function with ethereum compatibility", *Cryptology ePrint Archive*, 2024. [Online]. Available: https://eprint.iacr.org/2024/968.

[85] *Eip-214: New opcode staticcall*. Accessed: Nov. 18, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-214.

[86] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication", Feb. 1997. DOI: 10.17487/RFC2104. [Online]. Available: https://dl.acm.org/doi/abs/10.17487/RFC2104.

[87] *Next.js by vercel - the react framework*. Accessed: Nov. 16, 2023. [Online]. Available: https://nextjs.org/.

[88] D. P. Bauer, "Unit tests for smart contracts", *Getting Started with Ethereum*, pp. 49–53, 2022. DOI: 10.1007/978-1-4842-8045-4_4. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-8045-4_4.

APPENDIX

A

ELLIPTIC CURVE VALUES

List of elliptic curve values used in the various algorithms.

G2 Points:

$x_1 \leftarrow 10857046999023057135944570762232829481370756359578518086990519993285655852781$

$x_2 \leftarrow 11559732032986387107991004021392285783925812861821192530917403151452391805634$

$y_1 \leftarrow 8495653923123431417604973247489272438418190587263600148770280649306958101930$

$y_2 \leftarrow 4082367875863433681332203403145435568316851327593401208105741076214120093531$

Modulo q for G1 Point Negation:

$q \leftarrow 21888242871839275222246405745257275088696311157297823662689037894645226208583$

B

FRONTEND - NAVBAR COMPONENT

The complete render function for the `Navbar` component in the application, which handles user login/logout and displays dynamic content, is shown below.

```
return (
        <div className="flex flex-row justify-between items-end gap-[72px] max-md:flex-col
        ↪  max-md:text-center">
                <div className="text-6xl font-bold">AA Wallet</div>
                <div className="flex flex-row items-end gap-[12px] max-md:flex-col
                ↪  max-md:text-center">
                        {isLoggedIn ? (
                                <a
                                        href={`https://sepolia.etherscan.io/address/${scaAddress}`}
                                        target="_blank"
                                        className="btn text-white bg-gradient-1
                                        ↪  disabled:text-white transition ease-in-out
                                        ↪  duration-500 transform hover:scale-110
                                        ↪  max-md:w-full"
                                        >
                                                {username || "Logged In!"}
                                                </a>
                        ) : (
                        <button
                                disabled={isLoggingIn}
                                onClick={handleSocialLogin}
                                className="btn text-white bg-gradient-1 disabled:opacity-25
                                ↪  disabled:text-white transition ease-in-out duration-500
                                ↪  transform hover:scale-110 max-md:w-full"
                        >
                                {isLoggingIn ? "Logging In" : "Log In"} With Email
                                {isLoggingIn && (
                                        <span className="loading loading-spinner
                                        ↪  loading-md"></span>
                                                        )}
                        </button>
                                                )}
                                {isLoggedIn && (
                                        <button
                                        onClick={handleLogout}
                                        className="btn text-white bg-gradient-2
                                        ↪  disabled:opacity-25 disabled:text-white
                                        ↪  transition ease-in-out duration-500 transform
                                        ↪  hover:scale-110 max-md:w-full"
                                        >
```

```
30                                      {isLoggingOut ? "Logging Out" : "Log Out"}
31                                      {isLoggingOut && (
32                                              <span className="loading loading-spinner
                                                ↪  loading-md"></span>
33                                              )}
34                                      </button>
35                                      )}
36                                  </div>
37
38                                      {/* login pop-up modal */}
39                                      <dialog className={`modal ${isLoggingIn &&
                                        ↪  "modal-open"}`}>
40                                      <div className="modal-box flex flex-col gap-[12px]">
41                                      <h3 className="font-bold text-lg">Enter your
                                        ↪  email!</h3>
42                                      <input
43                                      placeholder="email"
44                                      onChange={onEmailChange}
45                                      className="input border border-solid border-white"
46                                      />
47                                      <div className="flex flex-row justify-end
                                        ↪  max-md:flex-col flex-wrap gap-[12px]">
48                                      <button
49                                      onClick={handleLogin}
50                                      className="btn bg-gradient-1 text-white transition
                                        ↪  ease-in-out duration-500 transform
                                        ↪  hover:scale-110"
51                                      >
52                                      Login
53                                      </button>
54                                      <button
55                                      onClick={closeModal}
56                                      className="btn bg-gradient-2 text-white transition
                                        ↪  ease-in-out duration-500 transform
                                        ↪  hover:scale-110"
57                                      >
58                                      Close
59                                      </button>
60                                  </div>
61                              </div>
62                          </dialog>
63                  </div>
64  );
65
```

C

FRONTEND - WALLET CONTEXT WRAPPER

The complete render function that allows the application to provide a Wallet at the root level, thus injecting the state throughout the whole application's pages.

```
return (
        <WalletContext.Provider
                value={{
                                login,
                                logout,
                                isLoggedIn,
                                provider,
                                ownerAddress,
                                scaAddress,
                                username,
                }}
        >
                {children}
        </WalletContext.Provider>
);
```

D

FRONTEND - MAGIC API KEY

Secret key used to access Magic's API. Part of if was omitted for obvious reasons.

```
NEXT_PUBLIC_MAGIC_API_KEY=pk_live_0ª7ª2C*AA2*1A77*
```

E

IDENTITY MANAGEMENT

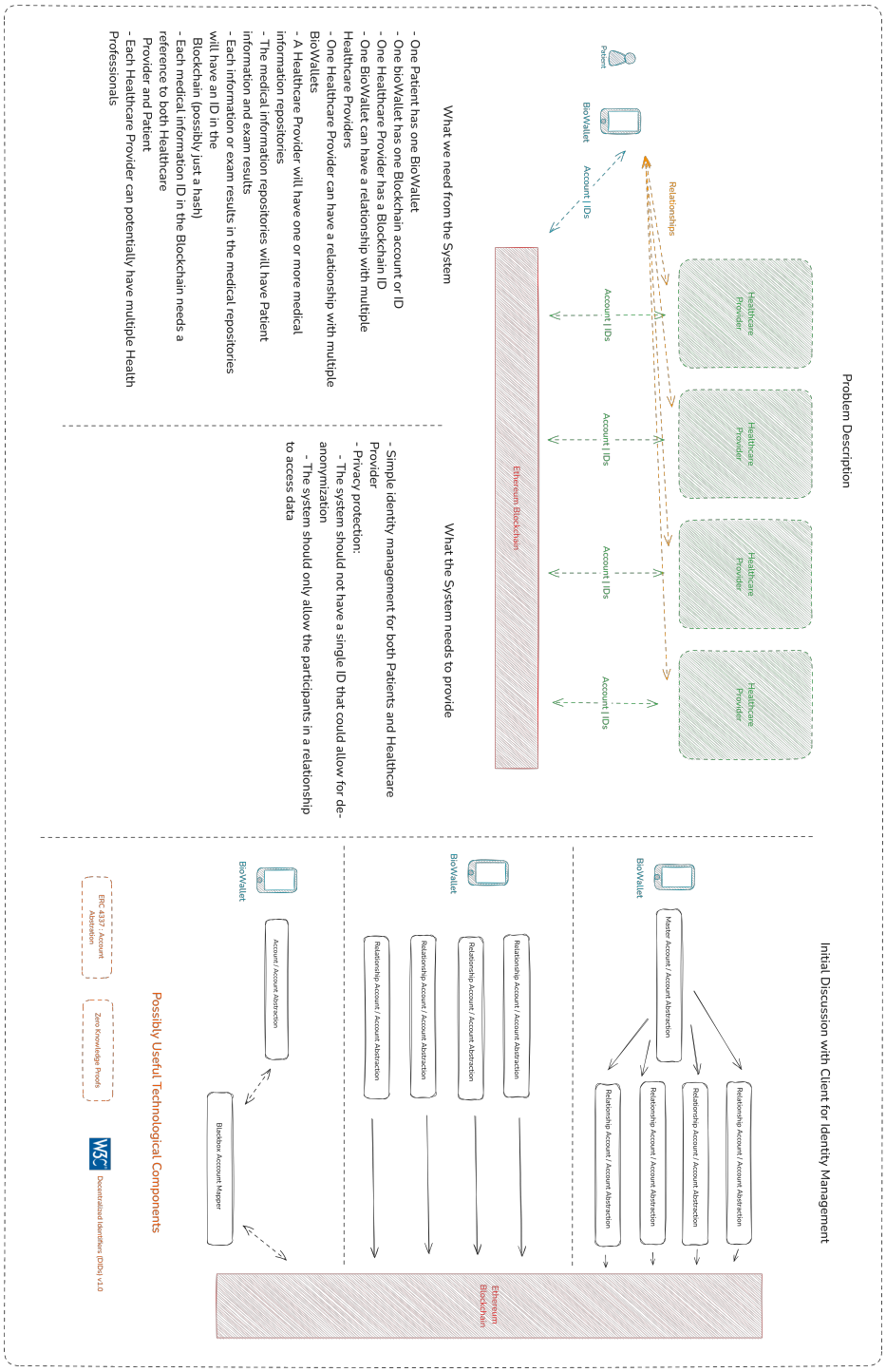


Figure 30: Identity Management

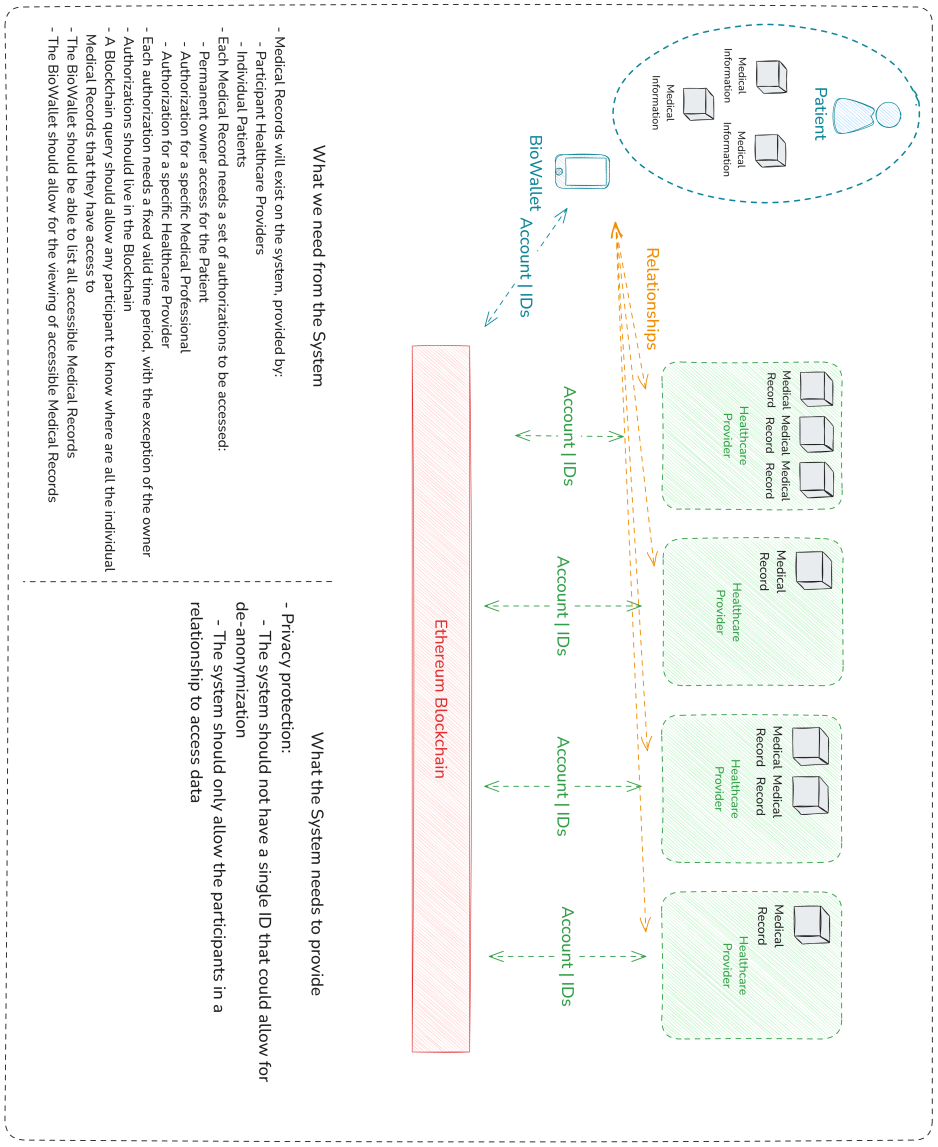


Figure 31: Access Management