Harmonizing People, Processes, and Technology for Robust Cybersecurity

D5.1 Secure Data Exchange Platform v1

Document Summary Information

Grant Agreement No	101158555	**Acronym**	CYberSynchrony
Full Title	Harmonizing People, Processes, and Technology for Robust Cybersecurity		
Start Date	01/06/2024	**Duration**	36 months
Project URL	https://www.cybersynchrony.eu/		
Deliverable	D5.1 Secure Data Exchange Platform v1		
Work Package	WP5 Cross-Border Cyber Collaboration and Secure Response (CROSS-CORE)		
Contractual due date	31/5/2025	**Actual submission date**	30/05/2025
Type	Other	**Dissemination Level**	Public
Lead Beneficiary	BYTE		
Responsible Author	George Karavokiros (BYTE)		
Contributions from	A.Psychas (MAG)		

Funded by the European Union

ECCC

Funded by the European Union under Grant Agreement no. 101158555 and supported by the European Cybersecurity Competence Centre (ECCC). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible for them.

Revision history

Version	Issue Date	Changes	Contributor(s)
0.1	15.03.2025	Initial deliverable structure	George Karavokiros (BYTE)
0.2	25.03.2025	First draft with content from BYTE and MAG	George Karavokiros (BYTE), A.Psychas (MAG)
0.3	01.04.2025	Integration of DLT section, updated architecture diagrams	George Karavokiros (BYTE)
0.4	22.04.2025	Completed content across all sections, ready for review	A.Psychas (MAG), George Karavokiros (BYTE)
0.5	05.05.2025	Peer review	Kitty Kioskli (TRUST)
0.6	08.05.2025	Peer review	S.Papagiannopoulos(UPRC)
0.7	23.05.2025	Final draft addressing comments from internal review	George Karavokiros (BYTE)
0.8	29.05.2025	Final quality check	Panayiotis Klitou (EBOS)
1.0	30.05.2025	Final version ready for submission	George Karavokiros (BYTE)

Disclaimer

The content of the document herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the CYberSynchrony consortium make no warranty of any kind with regard to this material including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the CYberSynchrony consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the CYberSynchrony consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© CYberSynchrony Consortium This Deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

[1] According to CYberSynchrony's Quality Assurance Process for reports:

- 3 months before Deliverable's Due Date: Table of Contents ready to be reviewed by WP leader/ Technical Manager/ Quality Manager
- 2 months before Deliverable's Due Date: 50% should be complete. Review by the Quality Manager (& Technical Manager for software).
- 1 month before Deliverable's Due Date:100% should be complete. Review by the Quality Manager and Peer Reviewers.
- M: Sent to PC for Submission to the EC after addressing all comments by Quality Manager and Peer Reviewers

Table of Contents

1 Executive Summary 8
2 Introduction 9
2.1 Mapping CYberSynchrony Outputs 9
2.2 Linkage to other project outputs 9
2.3 Deliverable Structure 10
3 Alignment with CYberSynchrony Architecture 12
4 Distributed Data Exchange Platform (DDEP) 14
4.1 Component Description 14
4.2 Architecture 14
4.2.1 Cloud data exchange and InterPlanetary File System 16
4.2.2 Zero Knowledge Proof Authorization Mechanism 17
4.2.3 Differential Privacy 19
4.3 Efficiency and Accessibility 21
4.4 Why This Solution Is State-of-the-Art 23
4.5 Technologies Used 24
5 Conclusions 25
6 References 26
Annex I: Distributed Data Exchange Platform 27
Healthcare Use Case 27
Implementation of the blockchain framework's frontend 27
Authentication 27
Components 27
Services 33
Implementation of the blockchain framework's backend 35
Backend API overview 35
APIS for admins 36
APIS for patients 37
APIs for doctors 38
APIs for researchers 38
Blockchain and IPFS for data exchange 39
Notification System and Emergency Feature 40
Security and Privacy 44
File Security 45

List of Figures

Figure 1: Alignment of Distributed Data Exchange Platform with CYberSynchrony components 12
Figure 2: High-level blockchain architecture 15
Figure 3: Flow for uploading secure data exchange through IPFS. 16
Figure 4: User authorization process for secure data access using zk-SNARKs, with client-side proof generation and zero-knowledge verification. 19
Figure 5: A high-level flow for HE and DP operations combined. 21
Figure 6: IPFS advantages. 21
Figure 7 getNotifications() method 28

© CYberSynchrony, 2024

Figure 8 Api service for getting patients' notifications 29
Figure 9 Method for marking notifications as read 29
Figure 10 Method for handling yes button clicks 30
Figure 11 Api service for changning status on requests 30
Figure 12 UML flow diagram for the notification system 31
Figure 13: RequestEmergencyAccessDoctor method for the emergency access functionality 32
Figure 14 Method for generating emergency access proof 33
Figure 15 Method for file downloading 33
Figure 16: API Service 34
Figure 17: Authentication Service 34
Figure 18: Authentication Interceptor 35
Figure 19: Overview of some APIs in Swagger 36
Figure 20 Interaction with /registerDoctor endpoint through swagger 36
Figure 21 POST /login 37
Figure 22: Grant access function 38
Figure 23: API for researchers reading statistical data 39
Figure 24: File is encrypted even after knowing the CID 39
Figure 25 An authorized doctor can download patients' files 40
Figure 26: Emergency access method 41
Figure 27 API for emergency access 42
Figure 28: NodeJS function implementation for marking notifications as read. 42
Figure 29: NodeJS function implementation for retrieving patients' notifications. 43
Figure 30: UML diagram for user authentication. 45
Figure 31: AES execution time for encrypting files of various sizes. 46
Figure 32: Total time for file encryption and IPFS upload. 46
Figure 33: Unified Framework for Differential Privacy, Homomorphic Encryption, and Blockchain in Privacy-Preserving Data Analysis 47
Figure 34: Synthetic Dataset Example for Privacy-Preserving Healthcare Data Analysis 48
Figure 35 Differentially private computation of mean, sum, and count using PyDP, protecting individual values while enabling aggregate insights. 49
Figure 36 Homomorphic encryption using Pyfhel library 50

© CYberSynchrony, 2024

List of Tables

Table 1: Adherence to CYberSynchrony's GA 9
Table 2: Links to other Project Outputs/Work 10
Table 3: Distributed Data Exchange Platform alignment with CYberSynchrony objectives. 14
Table 4: Technology stack. 24
Table 5: Summarized results for encryption and uploading times. 46

© CYberSynchrony, 2024

Acronyms and abbreviations

Abbreviation / Term	Description
AES	Advanced Encryption Standard – a symmetric-key algorithm used here to encrypt files before they are uploaded to IPFS.
API / REST API	Application Programming Interface that follows the REST architectural style; exposes CRUD endpoints for the web and mobile clients.
CID	*Content Identifier* – the cryptographic hash returned by IPFS that uniquely addresses a file (content-addressable storage).
CKKS	Cheon–Kim–Kim–Song homomorphic-encryption scheme that supports approximate arithmetic on encrypted real numbers.
CTI	Cyber Threat Intelligence
DDEP	Distributed Data Exchange Platform – the hybrid stack (Fabric + IPFS + MySQL) that underpins CYberSynchrony's data-sharing architecture.
DP / LDP	Differential Privacy / Local Differential Privacy – techniques that inject calibrated noise into data or queries to prevent re-identification.
Groth16	A highly efficient zk-SNARK proving system that generates very small proofs and fast verification keys; chosen for emergency-access proofs.
HE	Homomorphic Encryption – cryptography that allows computations on ciphertexts; results decrypt to the same value as if computed on plaintexts.
HLF (Fabric)	Hyperledger Fabric – permissioned blockchain used to store immutable audit logs and encryption-key metadata.
IPFS	InterPlanetary File System – a peer-to-peer, content-addressable network used to hold large encrypted files off-chain.
JWT	JSON Web Token – signed token that carries user identity and role claims, protecting API routes without server-side sessions.
LAP	Laplace mechanism – DP technique that adds Laplace-distributed noise proportional to function sensitivity (used for continuous data).
MySQL	Relational database that stores operational metadata (users, roles, access tables) for low-latency queries.
PoC	Proof of Concept – an early demonstrator that validates technical feasibility.
Poseidon	Hash function optimised for arithmetic circuits; used inside the zk-SNARK circuit to commit the user's secret key.
RPC	Remote Procedure Call – backend mechanism (e.g., ipfs add) that lets the server invoke commands on other services.
SPA	Single-Page Application – the Angular front end that loads once and then communicates with the backend via APIs.

© CYberSynchrony, 2024

UFT	Unalterable Forensic Techniques
WASM	WebAssembly – portable binary format; the zk-SNARK proof generator runs in the browser as a WASM module.
zk-SNARK	*Zero-Knowledge Succinct Non-Interactive Argument of Knowledge* – cryptographic proof that lets a user prove a statement without revealing the secret witness.

© CYberSynchrony, 2024

1 Executive Summary

Deliverable D5.1 introduces the first integrated release of the CYberSynchrony Secure Distributed Data Exchange Platform (DDEP) a modular, cloud-ready environment that enables semi-trusted organisations to exchange sensitive data in real time without compromising confidentiality, integrity, or auditability. This platform plays a central role in fulfilling the objectives of WP5 by delivering the technical foundation for secure, privacy-preserving cross-border information sharing between Member States.

The platform's core architecture integrates three complementary technologies: a **permissioned Hyperledger Fabric ledger** for tamper-evident event logging, **IPFS** for scalable off-chain file distribution, and **MySQL** for low-latency querying of operational metadata. This separation of responsibilities allows each layer to excel at its designated task, achieving greater flexibility and performance than monolithic or public-chain alternatives.

Security and privacy are enforced end-to-end. All API calls are authenticated using JSON Web Tokens, while sensitive or high-risk operations are gated behind **Groth16 zero-knowledge proofs**, generated in-browser via WebAssembly and verified server-side within milliseconds. Verification outcomes are recorded immutably on chain. For privacy-preserving analytics, the platform combines **Local Differential Privacy** with **CKKS homomorphic encryption**. Noise is introduced at the source before encryption, and aggregate computations run directly on encrypted vectors, ensuring that statistical insights can be derived without revealing sensitive inputs.

From an operational perspective, the entire solution is containerised via Docker Compose, fully instrumented for DevSecOps pipelines, and exposed through Swagger-documented REST APIs alongside a responsive Angular frontend. Features such as real-time notifications, role-based access management, and AES-encrypted IPFS storage provide data owners with fine-grained control over who accesses what, when, and how — with all decisions mediated and enforced through smart contracts.

Considering the nature of this deliverable, which is classified as an "Other" type, a **complete demonstrator is provided in Annex I**, showcasing the preliminary implementation and functional proof of the system's capabilities.

© CYberSynchrony, 2024

2 Introduction

2.1 Mapping CYberSynchrony Outputs

This section presents the CYberSynchrony's Grant Agreement (GA) commitments, as extracted from the formal deliverable and task description, in respect to their outputs and work to be performed.

Table 1: Adherence to CYberSynchrony's GA

Component no	Component title	Respective Document Chapter(s)	Justification
DELIVERABLE			
D5.1	D5.1 – Secure Data Exchange Platform v1	Chapter 1-4, Annex I	This deliverable presents the architecture and initial implementation of CYberSynchrony's Secure Data Exchange Platform. It introduces the foundational components enabling secure, verifiable, and privacy-preserving data sharing, forming a critical building block of the overall platform.
TASKS			
T5.1	T5.1 – Fortifying Cross-Border Data Exchange	Chapter 3, Annex I	The architecture leverages distributed ledger technology to ensure the integrity, confidentiality, and traceability of data shared across Member States. Encryption mechanisms and hash registration processes directly address the task's focus on uncompromised, trustworthy data exchange.
T5.2	T5.2 – Advancing Uniform Data Sharing Standards	Chapter 3	The deliverable incorporates standardized data formatting, sharing protocols, and API design, ensuring interoperability and consistency in cross-border data exchange as defined in T5.2.

2.2 Linkage to other project outputs

The design and development of the Secure Data Exchange Platform were informed by previous architectural work, complementary forensic mechanisms, and upcoming integration requirements across the CYberSynchrony

© CYberSynchrony, 2024

project. The table below outlines how this deliverable aligns with and supports key outputs from other work packages and tasks.

Table 2: Links to other Project Outputs/Work

WP Number	Task Number	Deliverable Number related	Content
WP2	T2.4 – CYberSynchrony Ecosystem Design	D2.1	The architectural principles and system-wide interoperability requirements defined in Task 2.4 guided the design of the Secure Data Exchange Platform. Its modular interfaces, data pipelines, and ledger integration are directly informed by the system-level architecture outlined in D2.1.
WP3	T3.2 – Adopting Unalterable Forensic Techniques	D3.1	The blockchain-enabled integrity validation mechanism developed in this deliverable complements the UFT process described in Task 3.2. Hashes registered via this platform serve as verifiable proofs for log data, ensuring forensic soundness and traceability.
WP8	T8.1 – CYberSynchrony Integration and Use Case Planning	D8.1	The components developed here will be integrated and validated within the CYberSynchrony platform through the system-wide efforts of Task 8.1. Their architecture was designed with integration-readiness in mind, ensuring seamless communication with other tools in future phases.

2.3 Deliverable Structure

This deliverable is structured into four main chapters and one annex. It presents the DDEP, outlining its architecture, implementation details, and role in the broader CYberSynchrony ecosystem. The document also addresses cross-WP alignment and system integration considerations.

- **Chapter 1: Introduction** Sets the context and scope of the deliverable. It outlines how this work aligns with other CYberSynchrony components, references related outputs from WP2, WP3, and WP8, and presents the structure of the document.
- **Chapter 2: Alignment with CYberSynchrony Architecture** Describes how the internal architecture and secure exchange processes developed in this deliverable integrate with the CYberSynchrony platform. Emphasis is placed on the design of interoperable interfaces, trust mechanisms, and data lifecycle integrity.
- **Chapter 3: Distributed Data Exchange Platform** Details the main component of the deliverable. It covers the platform's technical architecture, including the use of a permissioned distributed ledger (Hyperledger Fabric), differential privacy mechanisms, IPFS-based sharing, and cryptographic validation workflows. The section also discusses integration readiness, security features, and technologies used.

© CYberSynchrony, 2024

- **Chapter 4: Conclusions** Summarizes the key outcomes of the work, its alignment with project objectives, and the direction for future integration and validation activities.
- **Annex I: Distributed Data Exchange Platform – Proof of Concept and Implementation Details** Includes a detailed view of the preliminary implementation, including architecture diagrams, use case examples (e.g., healthcare), data flow descriptions, frontend/backend interactions, emergency access management, and security enforcement features.

© CYberSynchrony, 2024

3 Alignment with CYberSynchrony Architecture

The Secure DDEP presented in this deliverable is designed to serve as a trusted data pipeline within the broader CYberSynchrony ecosystem. Rather than depicting the entire project architecture, this section focuses specifically on the **internal component alignment and integration processes** that enable secure, verifiable, and privacy-preserving data exchange — a core objective of Work Package 5.

The figure below presents the architectural layout of the data exchange mechanism, showcasing how the components interact across both **on-premise** and **cloud-hosted** environments.

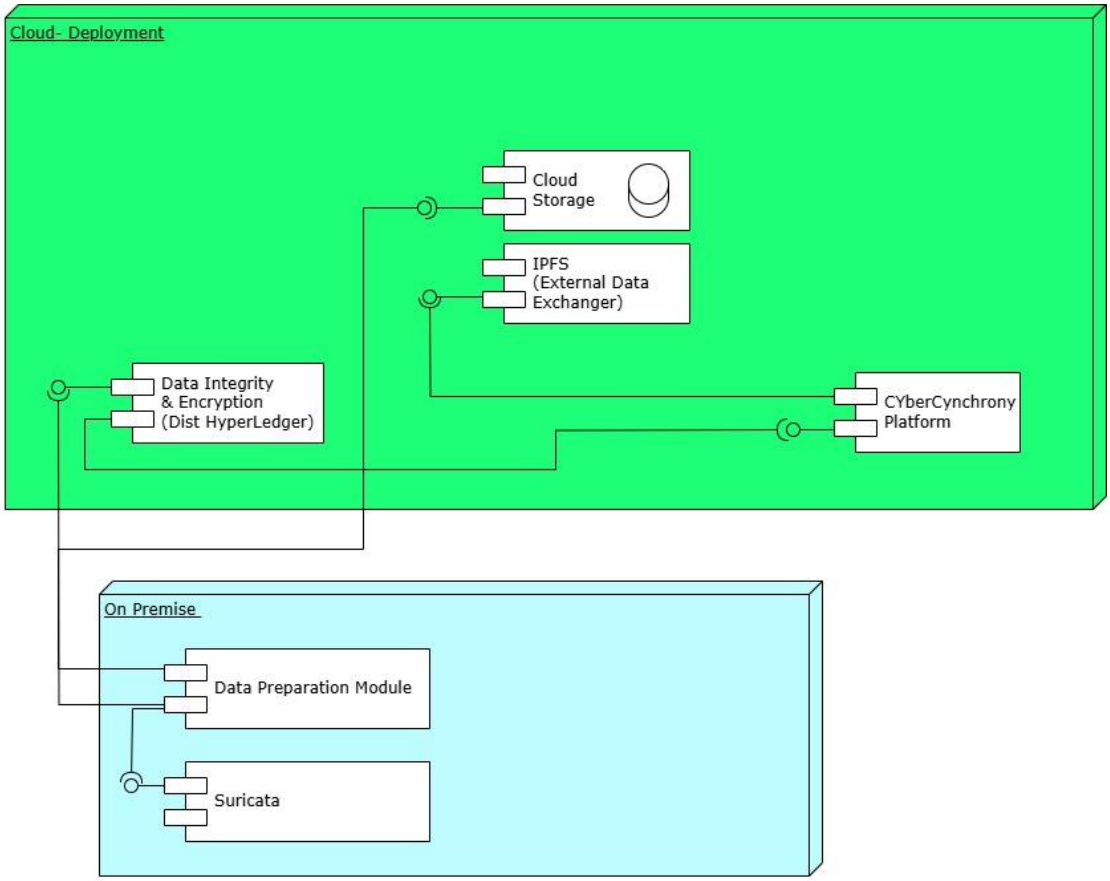


Figure 1: Alignment of Distributed Data Exchange Platform with CYberSynchrony components

At the **on-premise layer**, log data generated by tools such as **Suricata** is first collected and processed by the **Data Preparation Module (DPM)**. This Python-based module computes a **SHA-256 cryptographic hash** for each log batch and prepares it for submission to the platform.

The **hash is then registered on the Hyperledger-based Distributed Ledger**, ensuring tamper-evidence and providing a verifiable, timestamped record of each dataset. The original log batch is then securely uploaded to **Cloud Storage**, allowing authorized components of the CYberSynchrony platform to later access and verify its integrity by recomputing the hash and querying the ledger.

To facilitate external data sharing, the architecture also includes an **IPFS-based interface** (**Interplanetary File System**) labelled as an **External Data Exchanger**. This component supports secure and decentralized content distribution where needed, adding flexibility for integration scenarios involving third-party platforms or partner nodes.

© CYberSynchrony, 2024

Finally, the processed and validated data is exposed to the broader **CYberSynchrony Platform**, where it can be further analysed or correlated with threat intelligence, user behaviour, and decision support mechanisms.

This tightly integrated process reflects the goals defined in T5.1 and T5.2 by:

Ensuring data integrity and confidentiality through hashing, encryption, and ledger registration,

Enabling interoperability via structured exchange formats and standard APIs,

Supporting integration readiness for platform-wide adoption in WP8.

In alignment with deliverables (e.g., **D3.1 – Enhanced Threat Intelligence System** and **D4.1 – Integrated Cybersecurity Monitoring Suite**), the Secure Data Exchange Platform forms a critical link between **on-premise monitoring**, **cloud-based analysis**, and **cross-system data verification** — helping establish a modular and trustworthy cybersecurity architecture.

© CYberSynchrony, 2024

4 Distributed Data Exchange Platform (DDEP)

4.1 Component Description

The Secure Data Exchange Platform developed under WP5 represents a core trust-enabling mechanism in the CYberSynchrony architecture. It is built around a **permissioned blockchain infrastructure**, implemented using **Hyperledger Fabric**, to ensure that all data shared within the platform, especially log batches or forensic evidence, can be cryptographically verified for **integrity, origin, and immutability**. The component supports a **three-phase process**:

1. A log batch or data file is **hashed** locally using the SHA-256 algorithm via the **Data Preparation Module.**
2. The hash is then **submitted to the distributed ledger** (DLT) as a transaction, ensuring tamper-evident registration.
3. The corresponding data is **uploaded to Cloud Storage**, where it can later be accessed and verified by other platform components.

This approach ensures **forensic soundness** and **interoperability** in cross-border data exchange scenarios, enabling external components (e.g., IDS, CTI modules, forensic tools) to verify the authenticity of data without relying on a centralized trust anchor.

The platform was also designed with **extensibility and integration** in mind, supporting standardized API interfaces and optional decentralized storage via **IPFS** for external data distribution.

The following table outlines how the Secure Data Exchange Platform aligns with the core objectives of the CYberSynchrony project.

Table 3: Distributed Data Exchange Platform alignment with CYberSynchrony objectives.

Project Objective	Justification
Obj5 - Ensure data privacy and security during information exchange	The Secure Data Exchange Platform is built around a permissioned blockchain infrastructure that ensures all shared data is encrypted, integrity-verified, and auditable, directly addressing the objective of private and standardized cross-border communication.
Obj6 - Foster greater collaboration and knowledge sharing among stakeholders in the cybersecurity ecosystem	The platform enables secure, standardized data exchange between Member States and supports collaboration by incorporating MS as full participants in a federated Distributed Ledger Technology (DLT) network, allowing them to validate, share, and access information in a trusted environment.
Obj9 - Ensure a consistent and high level of security across all MS	By integrating Member States into the federated DLT infrastructure, the platform enforces a uniform, verifiable method for secure data exchange and trust anchoring, ensuring alignment with common security standards and supporting widespread adoption of harmonized practices across the EU.

4.2 Architecture

The initial implementation of this blockchain-based platform represents an important step toward creating a robust framework for secure communication and data exchange among multiple organizations. The system follows a **multi-layered architecture** that incorporates diverse technologies and interfaces, ensuring both strong data protection and convenient access for authorized parties, as illustrated in Figure 2.

© CYberSynchrony, 2024

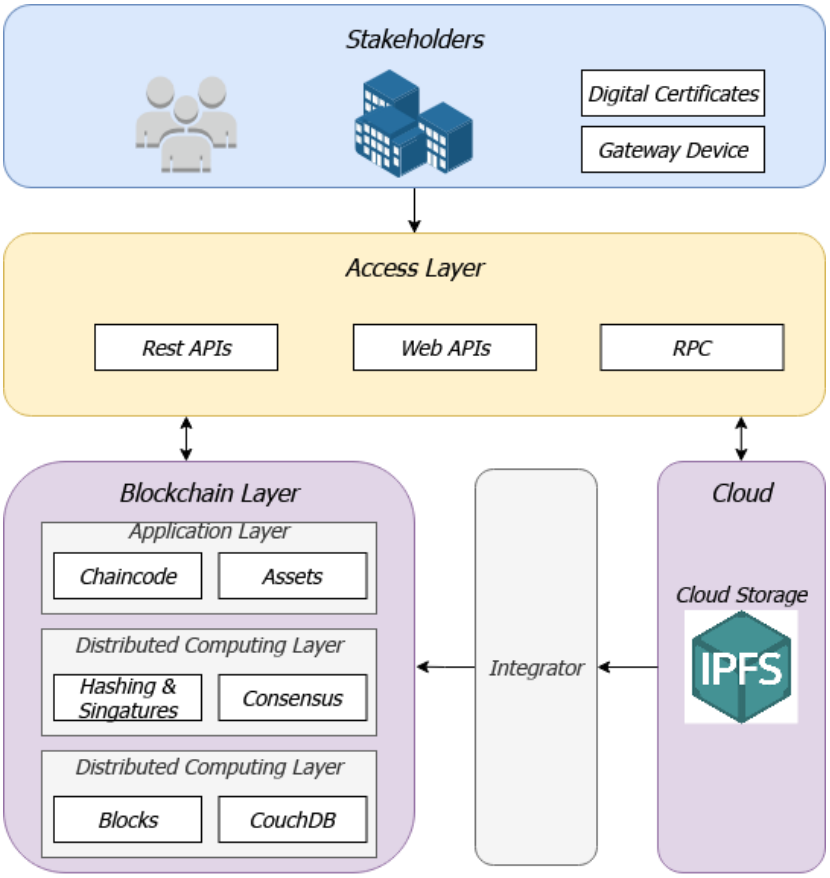


Figure 2: High-level blockchain architecture

1 **Stakeholder Layer**
At the top is the stakeholder layer, representing the framework's primary users. Depending on the use case, these can be employees, managers, clients, or other entities that need to access or share data. Interaction often involves digital certificates and secure gateway devices to authenticate users. Through this layer, stakeholders employ their devices (e.g., smartphones, laptops) to interact with the provided APIs in the Access Layer.

2 **Access Layer**
This layer contains critical technologies that manage user interactions and core system functions. Web APIs (e.g., RESTful APIs) handle client–server requests and ensure secure user authentication, with credentials and session information stored in a MySQL database. These APIs enable seamless web interactions and connectivity to blockchain services over the internet. A Remote Procedure Call (RPC) feature is employed to manage advanced tasks such as file uploading and downloading to IPFS.

3 **Blockchain Service**
The blockchain service is composed of several sub-layers:

- **Application Layer**
  Houses the chaincode (smart contracts), which define the rules governing transactions and interactions.
- **Distributed Computing Layer**
  Includes essential security mechanisms such as:
  - **Hashing:** Generates digital fingerprints for data integrity.
  - **Consensus:** Allows network participants to validate transactions collectively.

© CYberSynchrony, 2024

- **Digital Signature:** Ensures that transactions remain unaltered after issuance.

- **Storage Layer**
  Organizes data into blocks, which are linked together to form the blockchain. Additionally, a state database (e.g., CouchDB) records the current status of assets, facilitating queries and rapid data retrieval.

4 **Integrator**
Serving as middleware, the integrator bridges the Blockchain Service and a decentralized storage system (e.g., **IPFS**), which offloads large files or datasets from the blockchain itself. This design reduces on-chain storage overhead by leveraging a distributed, peer-to-peer network file system.

4.2.1 Cloud data exchange and InterPlanetary File System

The backbone of this blockchain-based framework is the integration of MySQL databases, IPFS (InterPlanetary File System) networks, and blockchain technology, ensuring a reliable and secure system for managing sensitive information. This approach leverages the strengths of traditional relational databases for metadata storage, coupled with modern decentralized networks and blockchain for enhanced file management and security.

At its core, the system uses MySQL as a relational database management system in combination with a private Hyperledger Fabric blockchain, where metadata about user data or files is stored. This metadata could include identifiers, timestamps (creation and modification), and references (links) to associated files stored over IPFS. Access is strictly regulated, requiring explicit permissions so that only authorized users can view or modify the metadata. This tight control safeguards data from unauthorized access and maintains high standards of privacy and security.

For larger files, the framework leverages the IPFS network. The InterPlanetary File System (IPFS) is a decentralized, peer-to-peer protocol and file storage system that enhances availability and fault tolerance by distributing content across a global network of nodes. Unlike traditional file storage, IPFS returns a unique content identifier (CID) that functions like a digital fingerprint**. This CID ensures data integrity, as any alteration to the file changes the CID, thus preventing undetected tampering**.

The process of adding a file to the IPFS network typically follows these steps (Figure 3):

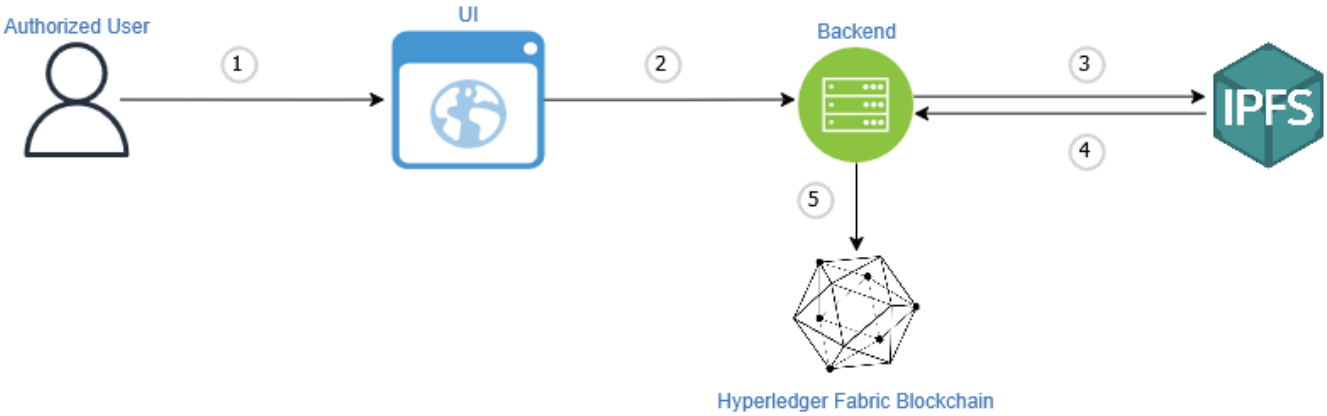


Figure 3: Flow for uploading secure data exchange through IPFS.

1. **User Authentication**: An authorized user accesses the application through a web-based user interface (UI). This confirms that only verified individuals can interact with and update records (Step 1).
2. **API Interaction for Record Updates**: After successful authentication, the backend grants the user the ability to interact with a set of system records via specialized APIs (Step 2). These interactions may include viewing or updating metadata, as well as uploading or retrieving files.

© CYberSynchrony, 2024

3. **Uploading a File to IPFS**: When the user uploads a file, the application calls the appropriate Remote Procedure Call (RPC) interface for an IPFS node (Step 3). By leveraging IPFS's distributed architecture, the file is replicated across multiple nodes to ensure redundancy and resilience.
4. **CID Generation and Retrieval**: Once the file is successfully added, IPFS generates a unique CID. This CID acts as the "key" required to access or download the file. Only users who possess the valid CID and have the necessary permissions can retrieve the file (Step 4).
5. **Blockchain Integration for Enhanced Security**: Finally, the CID is securely stored on the blockchain (Step 5). This step provides an immutable record of the file's reference and creates a transparent audit trail for tracking when and by whom the file is accessed or modified. Storing the CID on the blockchain also helps safeguard against unauthorized alterations.

This architecture exemplifies how traditional technologies (e.g., MySQL for relational data) can be combined with decentralized storage (IPFS) and blockchain to deliver a robust, efficient, and user-friendly data management solution. By capitalizing on the strengths of each technology layer—reliable metadata storage, distributed file capabilities, and immutable blockchain records—the framework addresses various challenges commonly found in digital data management while ensuring both security and scalability.

4.2.2 Zero Knowledge Proof Authorization Mechanism

In our framework, we developed a state-of-the-art proof-of-concept (PoC) demonstrator employing zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) in order to have a secure, privacy preserving approach for accessing data. Through zk-SNARKs, users can demonstrate their eligibility to access sensitive resources without exposing any private information. This method provides robust cryptographic assurances and protects the confidentiality of user-specific credentials.

Zero-knowledge proof [1],[2] is a cryptographic technique that can be used by an entity to prove that a statement is true without revealing any crucial information. zk-SNARKs is a zero-knowledge proof algorithm that requires no interaction between the sender and the verifier [3]. The main components of the zk-SNARK algorithm are the language and relations, the arithmetic circuits, and the quadratic arithmetic programs (QAPs).

Over the years, various zk-SNARK schemes have been introduced, each aiming to improve efficiency, scalability, and practicality in real-world applications. One of the earlier and influential implementations was *Pinocchio* [4], which laid the groundwork for practical verifiable computation using zk-SNARKs. Pinocchio introduced a highly optimized and succinct verification process, representing a significant leap in performance compared to earlier techniques. A core innovation of Pinocchio was its use of Quadratic Arithmetic Programs (QAPs) for representing computation circuits, a structure that remains foundational in zk-SNARK design today.

A more recent development is *Marlin*, a zk-SNARK protocol designed with scalability and flexibility in mind. Marlin introduces a universal and updatable setup, allowing the same trusted setup to be reused across different computations. This makes it particularly appealing for systems requiring flexibility and frequent updates. Additionally, it improves the efficiency of polynomial commitment schemes, resulting in smaller proofs and faster verification times. According to [5], Marlin has shown strong performance across several dimensions—such as proof size and generation time, when compared to other contemporary schemes like Mal19 and Sonic. However, despite its advantages, Marlin still falls short of outperforming the *Groth16* protocol in terms of overall efficiency.

In our work, we selected *Groth16* [6] as the foundation for the emergency authorization scheme. Groth16 is known for producing compact proofs and supporting extremely fast verification, which are critical features in time-sensitive and privacy-critical environments. Its low computational overhead and minimal proof size make it ideal for scenarios where decisions, such as granting or denying emergency access to sensitive data, need to be made quickly and securely. Moreover, its straightforward setup and mature tooling allow for seamless integration into web-based architectures, particularly when paired with client-side proof generation through

© CYberSynchrony, 2024

WebAssembly. For these reasons, Groth16 was the most suitable protocol to support our privacy-preserving, zero-knowledge–based emergency access control mechanism.

A central feature of the solution is a WebAssembly (WASM) file that performs zk-SNARK proof generation directly in a web browser or on the client side, eliminating any need for operating system-specific builds. By keeping proof generation local, sensitive information remains within the user's own environment. This design significantly reduces the risk of data exposure and enhances the overall trustworthiness of the authorization process.

The cryptographic commitments required within the zk-SNARK circuit are constructed using the Poseidon hash function, which is optimized for use within arithmetic circuits. Poseidon offers efficient and lightweight hashing, making it suitable for privacy-preserving protocols that must operate effectively even in constrained environments such as mobile devices, tablets, or embedded systems in edge deployments. A basic pseudocode that demonstrates the logic for our circuit is provided below.

```
FUNCTION EmergencyAccess(sk, pubHash):
    // sk: private input - secret key
    // pubHash: public input - pre-registered hash value of the secret key
    computedHash = PoseidonHash(sk)
    IF computedHash != pubHash THEN
        RAISE error "Secret key does not match public hash"
    ENDIF
    RETURN true
ENDFUNCTION
```

The system workflow (see Figure 4) begins when a user submits a request via the front-end interface. The WASM-based zk-SNARK proof generator runs directly within the browser, using the user's secret key to generate a zero-knowledge proof. This proof asserts that the private key corresponds to a pre-committed public hash value stored on the blockchain, without revealing the secret itself. The generated proof is then sent to the backend, where a verification key, retrieved from the IPFS, is used to validate the proof. If the verification succeeds, access is granted. This process occurs quickly and securely, upholding both privacy and access control integrity without relying on any centralized authority or exposing sensitive credentials.

This architecture demonstrates how privacy-preserving authorization can be achieved in real-time and under critical constraints. The combination of client-side proof generation, Groth16 efficiency, and Poseidon's lightweight hashing enables practical deployment in settings where both security and performance are paramount. Whether operating in a high-stakes operational environment or constrained edge conditions, the system ensures that authorized users can access the data they need instantly and securely, without compromising privacy.

© CYberSynchrony, 2024

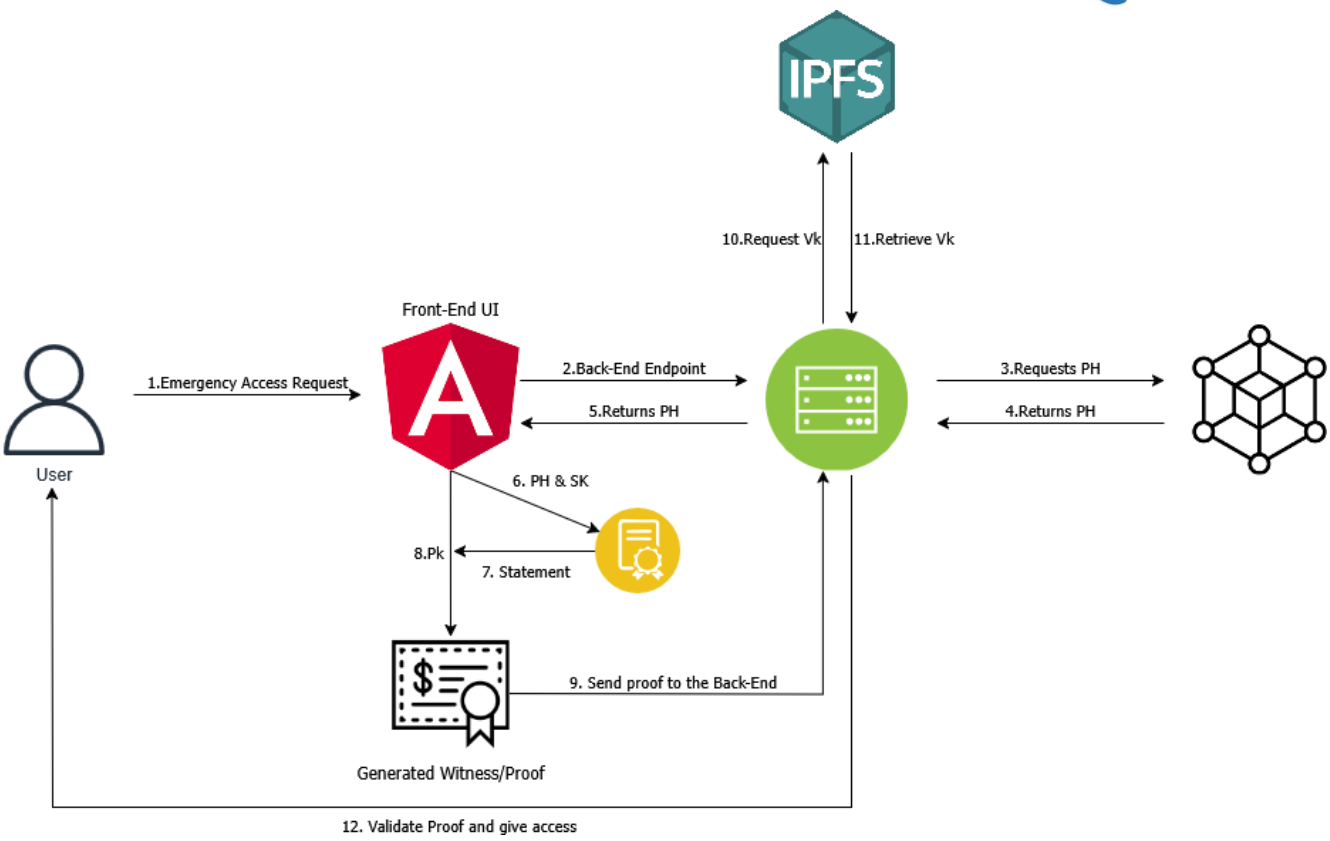


Figure 4: User authorization process for secure data access using zk-SNARKs, with client-side proof generation and zero-knowledge verification.

4.2.3 Differential Privacy

Sensitive data, whether generated by users, sensors, or systems, presents unique challenges for privacy protection. Traditional differential privacy methods, such as those applied during the early phases of development, commonly use a global differential privacy (GDP) approach by adding noise to aggregated outputs. While this technique protects against certain inference attacks, it often does so at the expense of individual-level privacy and data utility.

Under GDP, noise is typically added after computation, leaving the original data unchanged. This works well for centralised analytics but exposes vulnerabilities in distributed or federated systems, where individual records may be accessed or queried locally. Adversaries equipped with auxiliary knowledge may still be able to exploit unprotected raw data, even if final outputs are differentially private.

To close this gap the platform now adopts a **local differential-privacy (LDP) model implemented with the PyDP library**[1] (Google's open-source C++ DP stack wrapped for Python). PyDP injects calibrated noise into each value on the client side, so every record is already privatised when it reaches the first network hop.

For continuous attributes, the data can be perturbed using additive noise sampled from a Laplace distribution parameterized by ϵ (privacy budget) and the sensitivity Δf of the function being computed:

$$\hat{x} = x + \mathcal{L}\left(\frac{\Delta f}{\epsilon}\right),$$

[1] https://github.com/OpenMined/PyDP

© CYberSynchrony, 2024

where x is the original continuous value, $\hat{x}$ the perturbated, and $\mathcal{L}$, is the Laplace distribution generating noise. Δf is the sensitivity of the function (e.g., the maximum change in output resulting from the addition or removal of a single record). This ensures that continuous data remains usable while achieving differential privacy guarantees.

To strengthen privacy protections in distributed data environments, differential privacy and are combined into a unified pipeline (Figure 5). This hybrid approach ensures that send homomorphic encryption sensitive data is first anonymized through the injection of calibrated noise, and then encrypted in a way that allows computations to be performed directly on the encrypted data without requiring decryption at any stage of the processing workflow.

The process begins by applying local differential privacy mechanisms such as the Laplace-based perturbation directly to individual data records. This step obfuscates sensitive values while retaining statistical utility, protecting against re-identification even in the presence of background knowledge.

Once the data is differentially privatized, it is then secured using homomorphic encryption (HE) with **Pyfhel**[2] library. This cryptographic technique enables computations, such as aggregations, trends, and pattern detection to be performed directly on encrypted values. The results of these computations can later be decrypted by authorized entities using a private key, ensuring that raw data remains inaccessible throughout the process.

This two-layered design provides strong guarantees:

- Differential privacy protects individuals' data from being inferred even after decryption of results.
- Homomorphic encryption ensures that no intermediate system, analyst, or storage platform can access the unencrypted data.

Together, these techniques create a comprehensive framework for secure and privacy-preserving data processing.

Homomorphic encryption (HE) allows mathematical operations to be executed on ciphertexts in such a way that, once decrypted, the result matches what would have been obtained had the operations been performed on plaintext values. This property makes HE particularly valuable in privacy-sensitive domains where data must remain encrypted during computation.

The CKKS [7] (Cheon–Kim–Kim–Song) scheme is used to support approximate arithmetic over real and complex numbers. It is especially effective in scenarios involving numerical datasets, as it allows efficient packing of multiple values into a single ciphertext and supports vectorized operations.

The general process involves:

1. Public Key Encryption: Data is encrypted using a public key before transmission or storage.
2. Encrypted Computation: Authorized systems perform mathematical operations on encrypted data without ever decrypting it.
3. Private Key Decryption: Final results are decrypted using the corresponding private key, available only to authorized parties.

The secure use of HE relies on proper key distribution and access control:

- The public key is shared with data providers or collaborators, enabling encryption of data at the source.
- The private key is kept securely by a designated party, ensuring that only authorized users can decrypt computation results.

When combined with differential privacy:

[2] https://github.com/ibarrond/Pyfhel

© CYberSynchrony, 2024

- Public key encryption is applied after data is noise-injected, preserving the statistical structure while further securing raw values.
- Computations are carried out on encrypted, already-privatized data.
- The decrypted output retains the privacy guarantees of the differential privacy mechanism, ensuring compliance even if the final outputs are shared.

This flow eliminates the need to trust any single computation node or intermediary with access to unencrypted data, providing end-to-end privacy.

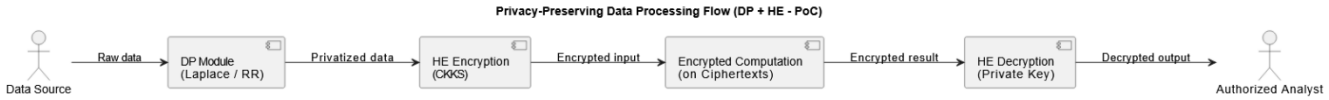


Figure 5: A high-level flow for HE and DP operations combined.

4.3 Efficiency and Accessibility

By adopting IPFS [8] within a blockchain-based architecture, several advantages arise (Figure 6), which are outlined as follows:

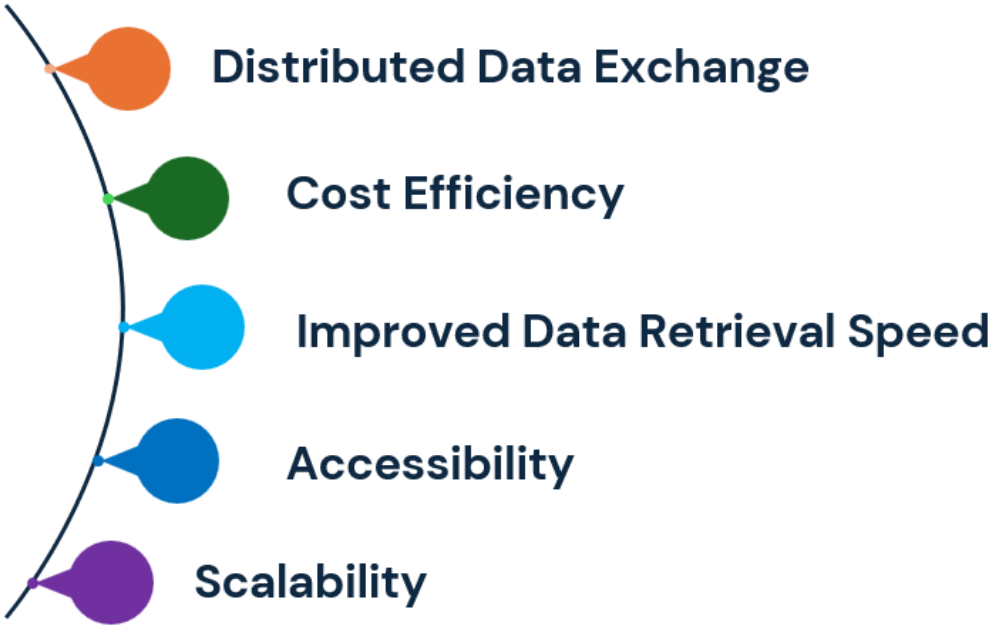


Figure 6: IPFS advantages.

© CYberSynchrony, 2024

Distributed Data Storage

IPFS operates through a global, distributed, peer-to-peer network in which the file system is stored across multiple nodes worldwide. Files, regardless of type or purpose, are not held in a single location. Instead, they are dispersed throughout the network:

- **Reduced Latency**: Each file request can be served by the node closest to the requester, significantly reducing data access time.
- **High Availability**: Files are replicated across several nodes. Even if part of the network fails, the data remains accessible, enhancing overall system resilience and ensuring continuous availability.

Cost Efficiency

Traditional centralized storage models often incur high costs related to redundancy and infrastructure scaling, especially when handling backups or increasing load. IPFS addresses these challenges through decentralized design:

- **Infrastructure Costs**: Instead of investing in centralized servers, stakeholders can leverage shared IPFS infrastructure for data exchange, reducing hardware and operational overhead.
- **Bandwidth Usage**: Since IPFS retrieves data from the nearest node, long-distance data transfers are minimized. This reduces bandwidth consumption and potentially lowers associated costs.

Improved Data Retrieval Speed

IPFS enhances retrieval speed using **content-addressable storage**, where each file is referenced by a unique cryptographic hash (Content Identifier - CID):

- **Efficient Caching**: Nodes cache accessed files locally. Subsequent access to these files is faster for users near the cached nodes.
- **Parallel Downloads**: Data can be retrieved from multiple nodes simultaneously, accelerating the process—especially beneficial when handling large datasets.

Accessibility

The design of IPFS improves accessibility in various ways:

- **Global Access:** IPFS distributes content across a peer-to-peer network rather than relying on centralized servers. This allows users to retrieve data from any node that stores it, enabling seamless access regardless of geographic location or server availability.
- **Content Addressing:** Files are identified by their cryptographic hash rather than their location. This ensures that the same content can be accessed consistently, regardless of which node hosts it, eliminating dependency on specific URLs or domains.
- **Offline Availability & Redundancy:** Since content is replicated across multiple nodes, files remain available even if individual nodes go offline. This enhances fault tolerance and ensures continuous availability during outages or targeted disruptions.
- **Interoperability:** IPFS uses standardized data addressing (Content Identifiers - CIDs) and retrieval protocols (e.g., libp2p, HTTP gateways), which improve integration with external systems and facilitate usage across heterogeneous environments, including browser clients, mobile devices, and cloud infrastructures.

Scalability

As data volumes continue to grow across all domains, underlying technologies must scale efficiently. IPFS provides native support for scalable data handling without a linear increase in cost or a drop in performance:

© CYberSynchrony, 2024

- **Horizontal Scaling**: The decentralized architecture allows seamless growth in storage capacity and network reach.
- **Support for Heterogeneous Data**: From structured records to sensor and IoT data, IPFS handles diverse data types efficiently, making it suitable for dynamic, data-intensive environments.

4.4 Why This Solution Is State-of-the-Art

A primary reason this hybrid framework stands out is its **balanced combination of MySQL, blockchain, and IPFS**, which together address longstanding challenges found in more conventional or single-technology solutions. Traditional databases excel at structured data management but often struggle with data integrity and transparency when it comes to auditing or preventing tampering. Public blockchains, while robust against tampering, typically face significant performance bottlenecks and high transaction costs, making them unsuitable for large-scale data storage. In contrast, this framework leverages a **private (permissioned) blockchain**, which maintains immutability and cryptographic assurance while offering better scalability, privacy, and control for enterprise applications. By selectively offloading bulk file storage to IPFS, the system lightens the on-chain burden, thereby enhancing throughput and scalability [9]. This modular design ensures that each component focuses on its specialized role, MySQL for relational data, the blockchain for cryptographic security and immutability, and IPFS for distributed file and data exchange, resulting in a synergy that is notably more performant and flexible than monolithic or purely decentralized architectures.

Additionally, the platform's analytics tier is state-of-the-art because it marries Local Differential Privacy with CKKS homomorphic encryption on top of a permissioned blockchain. Per-record noise is injected at the edge, shielding raw values even if a node is breached; the now-noised vectors are then re-encrypted, allowing aggregate queries and model updates to run directly on ciphertexts. Recent studies show that this "noise-then-encrypt" pipeline is the most effective way to preserve utility while blocking both reconstruction and ciphertext-inspection attacks [10], [11]. By anchoring every encrypted, differentially-private update to Fabric, our framework inherits the auditability these works call for, yet avoids their throughput bottlenecks through IPFS off-chain storage, placing it firmly at the forefront of privacy-enhancing data-exchange design.

Equally advanced is the **authorisation layer for critical situations, which replaces traditional credentials with Groth16-based zk-SNARK proofs verified on-chain**. Each access request carries a succinct proof, generated in the browser via WebAssembly, that the requester holds the requisite secret without revealing it [12]. Our implementation adopts the same Groth16 verification step and immediately commits the outcome (hash of proof + decision) to Fabric, giving an immutable, replay-resistant audit trail. By integrating client-side proof generation, millisecond-level server verification, and tamper-evident ledger logging, the platform delivers a privacy-preserving access-control mechanism that represents the cutting edge of blockchain security engineering.

Finally, **user-centric governance** sets this approach apart from more opaque platforms by enabling participants to exercise granular control and receive real-time notifications about data interactions. Traditional solutions may either focus on fully centralized oversight losing the benefits of transparency and distribution or adopt open public networks that can compromise privacy. By coupling a permissioned blockchain with an optional off-chain layer, the system retains both **privacy and decentralization**, mitigating single points of failure and enhancing fault tolerance. Overall, the combination of cryptographically guaranteed immutability, efficient off-chain scalability, and flexible integration with existing relational databases establishes this framework as a contemporary benchmark in secure, transparent, and high-performance data management.

© CYberSynchrony, 2024

4.5 Technologies Used

The proposed platform relies on a carefully curated technology stack that balances developer productivity, enterprise-grade security, and ease of deployment across the consortium. Table 4 lists only those libraries, frameworks, and infrastructure components that are essential in the production build; each entry is paired with a short explanation of the specific role it plays in the overall architecture.

Table 4: Technology stack.

Technology	Purpose
Javascript/Angular	Front-end framework used to build the SPA, handle client-side routing, forms, and UI logic.
Javascript/NodeJS	Runtime for the REST back-end and utility scripts.
Hyperledger Fabric (Test Network)	Consortium blockchain that stores immutable audit records (access decisions, emergency events).
Express.js	Core HTTP server & routing layer for all REST endpoints.
MySQL (mysql2 driver)	Relational store for operational data such as user accounts and access-control tables.
snarkjs	Generates and verifies Groth16 zk-SNARK proofs for emergency-access authorisation.
poseidon-lite	Browser- and server-side Poseidon hash function used to commit secrets in zero-knowledge circuits.
jsonwebtoken (JWT)	Issues and verifies stateless access tokens to protect API routes.
bcrypt	Secure password hashing for user credential storage.
Swagger / swagger-ui-express	Auto-generated, interactive API documentation exposed at /api-docs.
Multer	Middleware that validates and streams file uploads (e.g., patient documents).
CORS & Morgan	Security (cross-origin policy) and structured request logging, respectively.
Angular Material + Bootstrap	Component libraries that supply the application's responsive UI widgets and styling.
Docker (+ Compose)	Containerises the Angular front-end, Node.js back-end, Fabric peers/orderer, and MySQL for reproducible deployment.

© CYberSynchrony, 2024

5 Conclusions

Deliverable D5.1 presents the first integrated release of the DDEP, a core building block within CYberSynchrony's secure information sharing architecture. The platform combines a permissioned Hyperledger Fabric ledger with IPFS-based off-chain storage, a MySQL operational store, and an Angular frontend. All components are containerized and communicate securely via REST and RPC APIs, protected with JWT authentication and zero-knowledge proofs. This design enables semi-trusted entities to exchange sensitive data, manage access dynamically, and invoke emergency overrides—while preserving a tamper-evident, auditable record of all transactions.

The platform advances technical capabilities in multiple areas. Client-side Groth16 zk-SNARK proofs are compiled to WebAssembly, allowing operators to authenticate securely using break-glass credentials without revealing private material. Server-side verification is enforced, with all successful operations committed to the blockchain. Differential privacy is combined with additive homomorphic encryption, enabling parties to publish and aggregate statistical summaries without compromising underlying data. In addition, real-time notifications and a complete DevSecOps toolchain support rapid deployment and operational synchronisation across environments.

These innovations directly address several of CYberSynchrony's core objectives. The platform's use of encrypted communication, verifiable audit trails, and decentralized control mechanisms aligns with OBJ5, ensuring secure and privacy-preserving information exchange. Through its design as a federated distributed ledger, where participating Member States act as validating nodes, it supports OBJ6 by fostering secure collaboration and shared visibility across stakeholders. The standardized architecture, modular APIs, and policy-aligned enforcement mechanisms contribute to OBJ9, promoting a consistent and high-assurance cybersecurity posture across Member States.

In terms of threat modeling, the platform is designed to mitigate risks associated with **unauthorized access**, **data tampering**, **credential compromise**, and **insider misuse**. It assumes a powerful adversary capable of **intercepting, modifying, or injecting data**, including **compromised internal actors**. To counter these threats, the platform employs **zero-knowledge proofs** to protect sensitive authentication workflows, **cryptographic hashing with blockchain anchoring** for data integrity, **role-based access control** for scoped permissions, and **encrypted, audit-logged communication channels** to ensure traceability and non-repudiation. While a full formal threat model will be detailed in future security validation phases, the current design reflects a high-assurance baseline against tampering, privilege escalation, and data forgery.

Looking ahead, the next development steps will focus on integrating the DLT-based data exchange mechanisms into the project's defined use cases, enabling operational validation across diverse application domains. Particular emphasis will be placed on ensuring interoperability with existing CYberSynchrony components, including the UFT, CTI subsystems (e.g., MISP), and the monitoring suite presented in D3.1. These integrations will ensure that the data exchange platform functions not in isolation, but as a foundational service within a cohesive, modular cybersecurity framework.

© CYberSynchrony, 2024

6 References

[1] Fiege, U., Fiat, A., & Shamir, A. (1987, January). Zero knowledge proofs of identity. In Proceedings of the nineteenth annual ACM symposium on Theory of computing (pp. 210-217).

[2] Goldwasser, S., Micali, S., & Rackoff, C. (2019). The knowledge complexity of interactive proof-systems. In Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali (pp. 203-225).

[3] Chen, T., Lu, H., Kunpittaya, T., & Luo, A. (2022). A review of zk-snarks. arXiv preprint arXiv:2202.06877.

[4] B. Parno, J. Howell, C. Gentry and M. Raykova, "Pinocchio: Nearly Practical Verifiable Computation," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2013, pp. 238-252, doi: 10.1109/SP.2013.47.

[5] Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N. (2020). Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In: Canteaut, A., Ishai, Y. (eds) Advances in Cryptology – EUROCRYPT 2020. EUROCRYPT 2020. Lecture Notes in Computer Science(), vol 12105. Springer, Cham. https://doi.org/10.1007/978-3-030-45721-1_26

[6] Groth, J. (2016). On the Size of Pairing-Based Non-interactive Arguments. In: Fischlin, M., Coron, JS. (eds) Advances in Cryptology – EUROCRYPT 2016. EUROCRYPT 2016. Lecture Notes in Computer Science(), vol 9666. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-49896-5_11

[7] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. In Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23 (pp. 409-437). Springer International Publishing.

[8] Mahmud, M., Sohan, M.S.H., Reno, S. et al. Advancements in scalability of blockchain infrastructure through IPFS and dual blockchain methodology. J Supercomput 80, 8383–8405 (2024). https://doi.org/10.1007/s11227-023-05734-x

[9] Huang, D. C., Liu, L. C., Deng, Y. Y., Chen, C. L., & Zeng, K. W. (2024). A hyperledger fabric-based EMR sharing mechanisms with proxy re-encryption and IPFS. Cluster Computing, 27(5), 6953-6983.

[10] Firdaus, M., Larasati, H. T., & Hyune-Rhee, K. (2025). Blockchain-based federated learning with homomorphic encryption for privacy-preserving healthcare data sharing. Internet of Things, 101579.

[11] Tawfik, A.M., Al-Ahwal, A., Eldien, A.S.T. et al. PriCollabAnalysis: privacy-preserving healthcare collaborative analysis on blockchain using homomorphic encryption and secure multiparty computation. Cluster Comput 28, 191 (2025). https://doi.org/10.1007/s10586-024-04928-z

[12] Ma, S., Zhang, X. Integrating blockchain and ZK-ROLLUP for efficient healthcare data privacy protection system via IPFS. Sci Rep 14, 11746 (2024). https://doi.org/10.1038/s41598-024-62292-9

[13] Li, S., Xu, C., Zhang, Y., Du, Y., & Chen, K. (2022). Blockchain-based transparent integrity auditing and encrypted deduplication for cloud storage. IEEE Transactions on Services Computing, 16(1), 134-146.

[14] Molud Esmaili and Ken Christensen. 2025. Performance Modeling of Public Permissionless Blockchains: A Survey. ACM Comput. Surv. 57, 7, Article 174 (July 2025), 35 pages. https://doi.org/10.1145/3715094

[15] Kumar, S., Bharti, A. K., & Amin, R. (2021). Decentralized secure storage of medical records using Blockchain and IPFS: A comparative analysis with future directions. Security and Privacy, 4(5), e162.

[16] Azbeg, K., Ouchetto, O., & Andaloussi, S. J. (2022). BlockMedCare: A healthcare system based on IoT, Blockchain and IPFS for data management security. Egyptian informatics journal, 23(2), 329-343.

[17] Cruz, J. P., Kaji, Y., & Yanai, N. (2018). RBAC-SC: Role-based access control using smart contract. Ieee Access, 6, 12240-12251.

© CYberSynchrony, 2024

Annex I: Distributed Data Exchange Platform

Healthcare Use Case

Although this framework can be applied to a variety of secure data-sharing scenarios, it is particularly well-suited for healthcare environments, where stringent privacy and access control are paramount. Sensitive patient records require robust protections and clear auditability—elements that the framework delivers by combining a blockchain ledger, decentralized file share, encryption, and role-based APIs. The following sections outline how the frontend, developed in Angular, implements these measures to facilitate real-time notifications, granular access management, and an emergency authorization workflow for healthcare professionals.

Implementation of the blockchain framework's frontend

The frontend has been developed using the Angular framework, adhering strictly to best practices and security policies to guarantee a safe and secure user experience. In the following sections, we delve into and thoroughly examine the key principles underlying this software's design and implementation.

Authentication

In the frontend developed with Angular, a structural directive `*ngIf` is used for toggling the login form with the logged-in user interface, according to the **isLogin** Boolean flag. When **isLogin** is false, meaning the user is not logged in, it will show the form to enter username and password. This form utilizes Angular's template-driven forms approach, represented by `ngForm`, capturing and processing credentials securely provided by the user.

Authentication logic in the **LoginComponent** is orchestrated using properties such as isLogin, errorMessage, role, username, org, and the Boolean roleLoggedIn to manage authentication state and store user information. Key dependencies including **ApiService**, **AuthenticationService**, and routing are injected through the component's constructor, enabling API communication, authentication processes, and navigation management. A method **isUserLogin**() checks the login status by verifying the authentication token from local storage, and sets **roleLoggedIn** based on the user's role if the token is valid.

Upon submission of the login form, the form data is logged and sent to the login endpoint as a POST request using **ApiService**. If login is successful, the user's details are stored in local storage, and the user is redirected to the interface corresponding to their role. On failure, an "Invalid Username or Password" message is displayed to notify the user. The **isUserLogin**() function also supports conditional checks for login status by verifying the token. A **logout**() method ensures users can securely sign out by clearing the local storage and refreshing the application. Lastly, there is a redirect function that facilitates post-login navigation; it logs the user's role and is intended to enable navigation.

A UML sequence diagram that represents the authentication process in detail is presented in Figure 12. The whole approach within authentication in the platform underlines robust, secure, and user-friendly login experience, which would use the capabilities of Angular to make users' credentials processed in safe hands at the same time, allowing for dynamic updates of the user interface depending on the authentication state.

Components

We have developed a different component for each action that an entity can conduct in the framework. In particularly there is a register component for admins, a read component for doctors and patients (to be able to read the medical record), a grant access component (for patients to give permissions to their doctors) etc.

Our application enforces stringent access control policies to ensure that each API endpoint can only be accessed by authorized identities. For instance, the endpoint to grant access to a doctor is exclusively available to patients, not doctors or researchers. We achieve this by leveraging conditional rendering in Angular. This is demonstrated in our template, where we use the `ngIf` directive to conditionally display content based on the user's role. If the user is not identified as a patient, the application prompts them to log in as a patient. Conversely, if the user

© CYberSynchrony, 2024

is authenticated as a patient, they are presented with the option to grant access to a doctor. This role-based access control is further supported by backend validations that prevent unauthorized API calls, ensuring that each action within our application is securely managed according to the user's authenticated role.

We have also included **notification** and **emergency access** components. These features bolster communication and access management capabilities, ensuring that doctors and patients have seamless interaction within the system, particularly during critical situations.

```
getNotifications() {
  if (this.isDoctor) {
    const doctorIdMatch = this.doctorUserName.match(/^doctor(\d+)$/i);
    const doctorId = doctorIdMatch ? parseInt(doctorIdMatch[1], 10) : null;
    const doctorObj = {
      doctorId: doctorId,
      username: this.doctorUserName,
    };

    this.apiService.getNotifications(doctorObj).subscribe(
      (response) => {
        if (response.notifications) {
          this.notifications = response.notifications.filter(
            (notification) => notification.status === 0 || notification.status === 1
          );
          console.log('Doctor Notifications:', this.notifications);

          this.updateUnreadNotificationsCount();
        } else {
          console.log('No notifications found for doctor.');
          this.notifications = [];
          this.unreadNotificationsCount = 0;
        }
      },
      (error) => {
        console.error('Error fetching notifications for doctor:', error);
        this.notifications = [];
        this.unreadNotificationsCount = 0;
      }
    );
  } else if (this.isPatient) {
    const patientIdMatch = this.patientUserName.match(/^patient(\d+)$/i);
    const patientId = patientIdMatch ? parseInt(patientIdMatch[1], 10) : null;
    const patientObj = {
      patientId: patientId,
      username: this.patientUserName,
    };

    this.apiService.getPatientNotifications(patientObj).subscribe(
      (response) => {
        if (response.notifications) {
```

Figure 7 getNotifications() method

The **getNotifications()** method (Figure 7) runs as soon as the component is initialised and the user's role has been resolved by **AuthService**. If the user is a doctor, the method extracts the numeric identifier from the username, assembles a doctorObj with doctorId and username, and submits it to apiService.getNotifications(). When the backend responds, the component keeps notifications whose local **notification-status** is either **0 = unread** or **1 = read/seen**, writes them into the notifications array, and derives unreadNotificationsCount by counting only the items whose status is 0.

If the user is a patient, the flow is parallel: the component creates a patientObj, invokes apiService.getPatientNotifications() (Figure 8), coerces the response to an array (in case the backend returns a single object) and treats every entry as unread, so the unread count is simply notifications.length. Whenever the backend supplies no data, both the list and the counter are reset to empty and zero.

© CYberSynchrony, 2024

```
this.apiService.getPatientNotifications(patientObj).subscribe(
  (response) => {
    if (response.notifications) {
      if (Array.isArray(response.notifications)) {
        this.notifications = response.notifications;
      } else {
        this.notifications = [response.notifications];
      }
      console.log('Patient Notifications:', this.notifications);
      this.unreadNotificationsCount = this.notifications.length;
    } else {
      console.log('No notifications found for patient.');
      this.notifications = [];
      this.unreadNotificationsCount = 0;
    }
  },
  (error) => {
    console.error('Error fetching notifications for patient:', error);
    this.notifications = [];
    this.unreadNotificationsCount = 0;
  }
);
```

Figure 8 Api service for getting patients' notifications

Opening the notification panel toggles showNotifications and immediately calls **markNotificationsAsRead()** (Figure 9). For doctors, this routine flips each local item from status 0 to status 1 and sends the corresponding doctorObj to apiService.markNotificationsAsRead() so that the backend records the change. For patients, the backend does not track per-item read state, so the method merely zeroes the unread counter.

```
markNotificationsAsRead() {
  if (this.isDoctor) {
    this.notifications.forEach((notification) => {
      if (notification.status === 0) {
        notification.status = 1;
      }
    });
    this.unreadNotificationsCount = 0;
    const doctorIdMatch = this.doctorUserName.match(/^doctor(\d+)$/i);
    const doctorId = doctorIdMatch ? parseInt(doctorIdMatch[1], 10) : null;
    const doctorObj = {
      doctorId: doctorId,
      username: this.doctorUserName,
    };
    this.apiService.markNotificationsAsRead(doctorObj).subscribe(
      (response) => {
        console.log('Doctor notifications marked as read.');
      },
      (error) => {
        console.error('Error marking notifications as read for doctor:', error);
      }
    );
  } else if (this.isPatient) {
    console.log('Patient notifications viewed.');
    this.unreadNotificationsCount = 0;
  }
}
```

Figure 9 Method for marking notifications as read

Patient decisions are captured by **onYesClick()** (Figure 10) and **onNoClick()**. Once either button is pressed the component sets actionTaken = true to disable further clicks, extracts the doctor's numeric identifier from the message text, and builds a userObj containing the notification ID, the patient's response ("yes" or "no"), both user IDs, and the patient's username.

© CYberSynchrony, 2024

```
onYesClick(notification: any) {
  console.log('Yes clicked for notification:', notification);
  notification.actionTaken = true;
  const doctorIdMatch = notification.message.match(/Doctor (\d+)/);
  const doctorId = doctorIdMatch ? parseInt(doctorIdMatch[1], 10) : null;
  const userObj = {
    notificationId: notification.id,
    response: 'yes',
    doctorId: doctorId,
    patientId: parseInt(this.patientUserName.replace(/^patient(\d+)$/i, '$1'), 10),
    username: this.patientUserName
  };
  this.apiService.changeStatus(userObj).subscribe(
    (response) => {
      console.log('Access request approved:', response);
      this.notifications = this.notifications.filter((n) => n.id !== notification.id);
      this.updateUnreadNotificationsCount();
    },
    (error) => {
      console.error('Error approving access request:', error);
      notification.actionTaken = false;
    }
  );
}
```

Figure 10 Method for handling yes button clicks

This object is sent to apiService.changeStatus() (Figure 11). If the server acknowledges success the processed item is removed from notifications and the unread count is recomputed; if an error occurs the button is re-enabled so the user can retry. In the backend the **access-request status**—a separate field from the notification-status discussed above—travels through three states: **"Pending"** until the patient acts, **"Approved"** after a "Yes," or **"Denied"** after a "No."

```
this.apiService.changeStatus(userObj).subscribe(
  (response) => {
    console.log('Access request denied:', response);
    this.notifications = this.notifications.filter((n) => n.id !== notification.id);
    this.updateUnreadNotificationsCount();
  },
  (error) => {
    console.error('Error denying access request:', error);
    notification.actionTaken = false;
  }
);
}
```

Figure 11 Api service for changning status on requests

A UML diagram, presenting the entire flow for the notification system is illustrated in Figure 12.

© CYberSynchrony, 2024

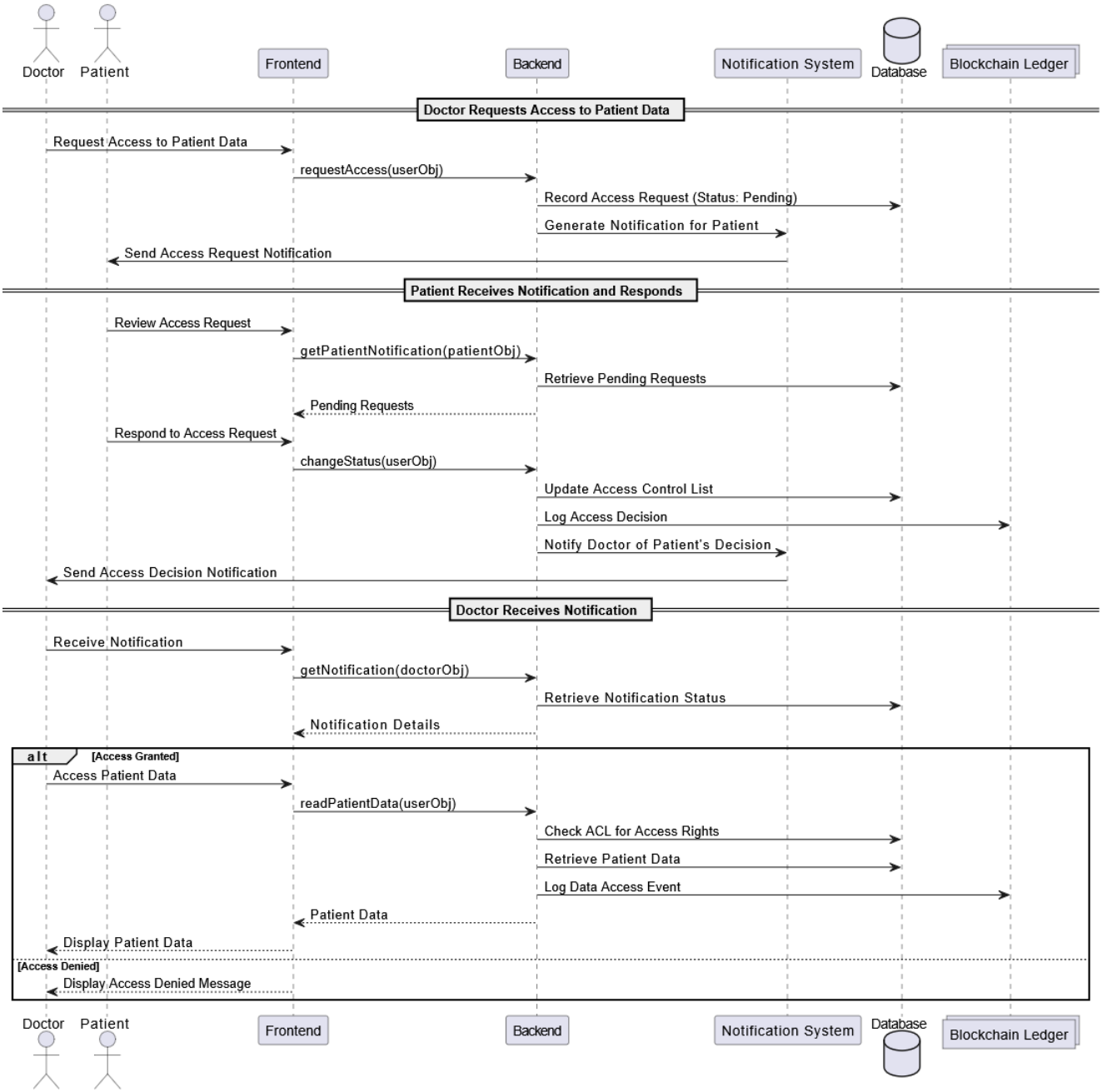


Figure 12 UML flow diagram for the notification system

```
async RequestEmergencyAccessDoctor(form: NgForm) {
  this.loading = true;
  const selectedPatientId = form.value.newPat;
  const selectedReason = this.selectedReason;
  try {
    await this.generateEmergencyProof();
    const payload = {
      username: this.username,
      org: this.org,
      patientId: selectedPatientId,
      reason: selectedReason,
      proof: this.emergencyProof,
      publicSignals: this.emergencyPublicSignals
    };
    console.log('Payload with emergency proof:', payload);
    this._api.postTypeRequest('emergencyAccessPatientRecord', payload).subscribe((res: any) => {
      this.loading = false;
      if (res) {
        this.patient = Array.of(res);
        console.log(res);
        this.returnsData = true;
        alert('Access granted successfully!');
      } else {
        console.log(res);
        alert(res);
        window.location.reload();
      }
    }, err => {
      this.loading = false;
      this.errorMessage = err['error'].message;
    });

  } catch (error) {
    this.loading = false;
    console.error('Error processing request:', error);
    alert('Error processing the request. Please try again.');
  }
}
```

Figure 13: RequestEmergencyAccessDoctor method for the emergency access functionality

The code snippet in Figure 13 belongs to GrantEmergencyAccessPatientComponent, the Angular page a doctor uses when an exceptional, time-critical situation requires access to a patient's record. When RequestEmergencyAccessDoctor() is invoked—typically by pressing a **"Grant emergency access"** button—the method first toggles the visual *loading* flag and extracts two pieces of user input from the template-driven form: the patient selected in the dropdown (newPat) and a clinically relevant justification chosen from reasonsList.

Before any network traffic is sent, the component calls generateEmergencyProof(). This helper (Figure 14) assembles a circuit input object containing the doctor's emergency secret key (sk), a fixed *flag* set to "1" (signalling an emergency path in the circuit), and an additional public secret chosen by the doctor. It then runs groth16.fullProve() against the WebAssembly binary **EmergencyAccess.wasm** and proving key **EmergencyAccess_final.zkey**—both shipped with the client. The browser therefore produces a Groth16 zk-SNARK proof and its accompanying public signals entirely on the client side; no private key material ever leaves the doctor's device.

© CYberSynchrony, 2024

```
async generateEmergencyProof(): Promise<void> {
  const emergencyInput = {
    sk: this.emergencySecret,
    flag: '1',
    pubSecret: this.publicSecret
  };
  try {
    const result = await groth16.fullProve(
      emergencyInput,
      'assets/EmergencyAccess.wasm',
      'assets/EmergencyAccess_final.zkey'
    );
    this.emergencyProof = result.proof;
    this.emergencyPublicSignals = result.publicSignals;
    console.log('Emergency proof generated:', result);
  } catch (error) {
    console.error('Error generating emergency proof:', error);
    throw error;
  }
}
```

Figure 14 Method for generating emergency access proof

When the promise resolves, the method builds a JSON payload and posts it to the /emergencyAccessPatientRecord endpoint through ApiService.postTypeRequest. If the backend validates the proof successfully, the UI records the returned object in patient, drops the spinner, sets returnsData to true, and pops a *"Access granted successfully!"* alert. Any falsy response, or a transport or server error, is caught: the spinner is cleared, the problem is logged, a generic alert is shown, and the component reloads to a clean state.

Services

Several services have been implemented to manage different aspects of the application in the framework, including API communication, user authentication, and file handling.

```
downloadFile(fileName: string): void {
  this.downloadloading = true;
  const payload = {
    fileName: fileName,
    role: this.role,
    patientId: this.patientIdTest,
    username: this.username,
    org: this.org_name
  };
  console.log(payload)
  const url = 'downloadFile';
  this._api.downloadFileRequest(url, payload).subscribe(
    (blob: Blob) => {
      this.downloadloading = false;
      saveAs(blob, fileName.split('/').pop() || 'downloadedFile.pdf');
    },
    error => {
      this.downloadloading = false;
      console.error('Download error:', error);
      alert('Failed to download file.');
    }
  );
}
```

Figure 15 Method for file downloading

The **APIService** (Figure 16) is designed to handle HTTP requests to a backend server. This service includes methods to perform GET, PUT, and POST requests. The method **downloadFile** (Figure 15) is specifically for downloading files and expects a response in the form of a blob through the **downloadFileRequest**. The service is tested to ensure it is correctly instantiated by Angular's dependency injection system, confirming its availability throughout the application.

© CYberSynchrony, 2024

```
export class ApiService {
  baseUrl = '[redacted]';
  constructor(private _http: HttpClient) {}
    getTypeRequest(url) {
    return this._http.get(`${this.baseUrl}${url}`).pipe(map(res => {return res;}));
    }
    putTypeRequest(url, payload) {
    return this._http.put(`${this.baseUrl}${url}`, payload).pipe(map(res => {return res;}));}

    postTypeRequest<T>(url: string, payload: any, responseType: 'json' | 'blob' | 'arraybuffer' = 'json'): Observable<T> {
      return this._http.post<T>(`${this.baseUrl}${url}`, payload, { responseType: responseType as any });
    }
    // Example usage for a file download
    downloadFileRequest(url: string, payload: (parameter) payload: any
      return this.postTypeRequest<Blob>(url, payload, 'blob');
    }
```

Figure 16: API Service

The **AuthService** manages the authentication state and storage of user data. It offers functionality to retrieve and set data in local storage, check user roles (such as admin, doctor, researcher, or patient), and manage authentication tokens (Figure 17). Each method checks for specific user roles based on data stored in local storage, helping to control access to various parts of the application. The service is also tested for proper instantiation, ensuring that it can be relied upon for authentication-related tasks.

```
@Injectable(
  {providedIn: 'root'}
  )
export class AuthService {
    public isAdmin : boolean = false;
    public isDoctor : boolean = false;
    public isPatient : boolean = false;
    public isResearcher : boolean = false;
  constructor(private http : HttpClient) { }
  getUserDetails(variableName) {
  return localStorage.getItem(variableName);
  }
  setDataInLocalStorage(variableName : string, data : string) {
    localStorage.setItem(variableName, data);
  }
  getToken() {
    return localStorage.getItem('token');
  }
  clearStorage() {
    localStorage.clear();
  }
  isUserDoctor(){
    if(this.getUserDetails("userData")=="doctor"){
      return true;
    }
    else {
      return false;
    }
```

Figure 17: Authentication Service

The **AuthInterceptor** (Figure 18) acts as an HTTP interceptor, modifying outgoing requests to include an authentication token in the header. This is critical for ensuring that requests to secure endpoints are properly authenticated. The interceptor is also part of the testing suite to verify its creation and integration into the HTTP handling pipeline.

Finally, the **FileService** is focused on file operations, specifically listing patient files. It makes a GET request to a backend endpoint designed to return patient-related files, enabling the application to handle file data efficiently. Like the other services, it is tested to confirm its creation and readiness to be used within the application. Each of these services is crucial for the robust functionality of the Angular application, handling everything from user

© CYberSynchrony, 2024

authentication to complex API requests and file management. All services are globally available due to their declaration with `@Injectable({providedIn: 'root'})`, which simplifies the management of these services across the entire application.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(public auth : AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>>{
    const idToken =localStorage.getItem("token");

    if (idToken){
      const cloned = request.clone({
        headers: request.headers.set("Authorization",
        "Bearer "+idToken)
      });
      return next.handle(cloned);
    }
    else{
    return next.handle(request);
  }
  }
}
```

Figure 18: Authentication Interceptor

Implementation of the blockchain framework's backend

Backend API overview

Our framework offers a robust backend framework designed to manage the registration of doctors, patients, and researchers efficiently, with each entity being securely integrated into a blockchain system. This integration not only ensures the secure collection and storage of critical data but also assigns unique digital IDs to each registrant, facilitating secure and traceable interactions with the system's ledger. For patients, the framework provides enhanced control over their medical records, allowing them to manage who has access to their sensitive information. Doctors and researchers gain the ability to access and update patient data, operating under strict compliance and privacy protocols. By leveraging blockchain technology, our solution ensures the highest levels of data integrity, security, and regulatory compliance, positioning it as a powerful tool for modernizing healthcare administration and improving patient data management. Some indicative APIs presented in Figure 19 using Swagger[3].

[3] https://swagger.io/

© CYberSynchrony, 2024

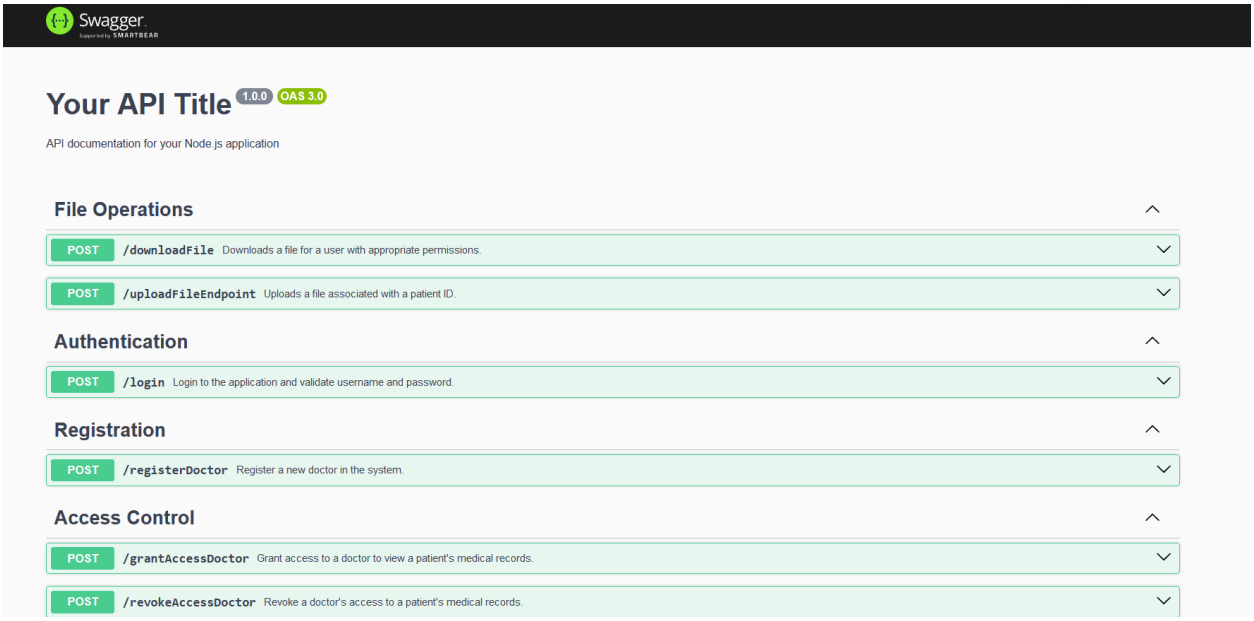


Figure 19: Overview of some APIs in Swagger

APIS for admins

There are three main endpoints that a hospital admin can interact with regarding the registration of the entities. In particularly there is one endpoint for registering doctors, one for patients and one for researchers. Each entity also is enrolled into the blockchain and gets a digital wallet (private/public key pair) in order to be able to interact and submit transactions later with the ledger.

The main purpose of the **/registerDoctor** endpoint is to add a new doctor to the system. This would involve registration and, hence, collection of all major details pertaining to the doctor, which includes name, specialty, and contact details. This information is then saved into the database, where the doctor is given access to the system with roles and permissions. In our framework, this is an operation that an admin user must do, and thus it entails control over who can operate inside the system as a healthcare provider. In Figure 20, we try to interact with this specific endpoint through the swagger.

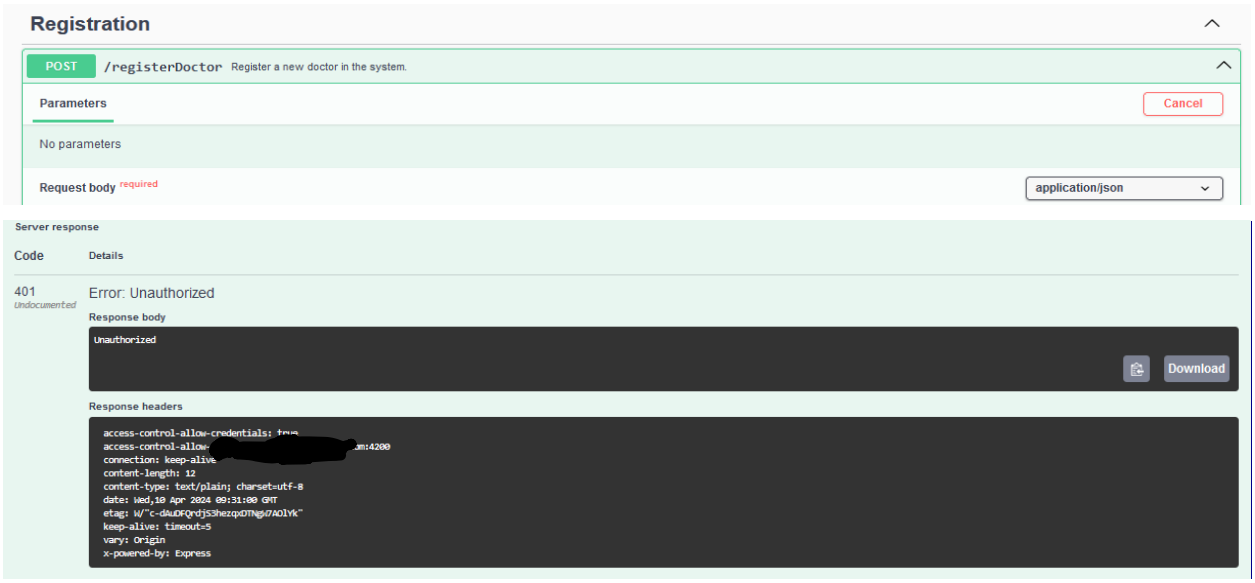


Figure 20 Interaction with /registerDoctor endpoint through swagger

© CYberSynchrony, 2024

The **/registerPatient** endpoint is essential in the digital onboarding of new patients. This includes such basic information as name, and contact details of the patients, which hence provide a basis for a digital record of the patient in the system. Upon submission, the data undergoes validation for completion and compliance with system requirements to ensure integrity and uniqueness of each patient's profile. Then, a new record in the database about the new patient is created, but only safely made and labelled by any unique label for easy reference and control.

Finally, the **/registerResearcher** endpoint can be used to register a researcher. This includes basic information as name and contact details. Researchers have access to anonymized and differential-private data for statistical analysis.

APIS for patients

Patients can interact with three endpoints. The first two are associated with access control policies. More specifically, patient can grant or revoke access to doctors according to their will. The **`/grantAccessDoctor`** API that allows the patients to selectively give permission to only those doctors or health providers of his choice to let them read or modify his health records. This ensures collaborative patient care in such a way that the communication and data sharing with the patient and the selected medical professional, are easily established. If allowed, the agreement by the patient enables the doctor to obtain access to history, results of health tests, and other important kinds of information that are vital for the diagnosis and treatment planning. On the other hand, the **/revokeAccessDoctor** endpoint ensures that patients can restrict doctors from accessing their sensitive medical data. More specifically, patients can see which healthcare professionals have access to their medical records through the user interface and be able to select the ones that they want to restrict access from a dropdown list. The code for the /login POST request in the nodejs backend in presented in Figure 21.

```
clientApp.post('/grantAccessDoctor', authenticateJWT, async (req, res) => {
  const patientObj = new User(req.body)
  const response = await app.grantAccessDoctor(patientObj);
  if (response.error) {
    res.send(response.error);
  } else {
    res.send(response);
  }
});
```

Figure 21 POST /login

while part of the code for the **grantAccessDoctor()** function in which the database relations to give permissions to a doctor or revoke them (by a similar **revokeAccessDoctor**() function) have been declared, is illustrated in Figure 22.

© CYberSynchrony, 2024

```
  const patientId = parseInt(userObj.username.replace(/^patient(\d+)$/i, '$1'));
  const doctorId = parseInt(userObj.doctorId);
  const check_query = 'SELECT 1 FROM hospitalInfo.patientDoctorAccess WHERE patientId = ? AND doctorId = ? LIMIT 1';
  const check_result = await queryAsync(connection, check_query, [patientId, doctorId]);
  console.log(check_result);
  if (check_result.length > 0) {
    const query = 'UPDATE hospitalInfo.patientDoctorAccess SET doctorAccess = 1 WHERE patientId = ? AND doctorId = ?';
    const result = await queryAsync(connection, query, [patientId, doctorId]);
    return result;

  }
  else {
    const insertQuery = 'INSERT INTO hospitalInfo.patientDoctorAccess (patientId, doctorId, doctorAccess) VALUES (?, ?, 1)';
    const result = await queryAsync(connection, insertQuery, [patientId, doctorId]);
    return result;

  }

} catch (error) {
  console.error(`grantAccessDoctor() --> Failed to grant access: ${error}`);
  throw new Error(`Failed to grant access: ${error}`);
} finally {
  if (connection) {
    connection.release();
  }
}
```

Figure 22: Grant access function

APIs for doctors

Doctors who have been granted permission by their patients can read or update their medical data by interacting with the **/readPatientData** and **/updatePatientData** endpoints. The **/readPatientData** API provides the doctor with access to the medical record of a patient to get comprehensive review details about his health history, including diagnoses, treatments, results from tests, and notes made by other health professionals, downloading files etc. in order to make decisions at high-quality level in terms of treatment. The access through this API is limited, while strong privacy and security rules guarantee that only authorized doctors are able to browse through sensitive patient information. The **/updatePatientData** API empowers doctors to make changes or additions to a patient's medical record. This includes, updating the patient's condition and any newly added test results to the record by uploading the associated files etc. The updates are under strict compliance with healthcare regulations for the sake of patient privacy.

APIs for researchers

We have implemented the endpoint for authorized researchers to be able to read patients' data in a differential private manner (Figure 23).

© CYberSynchrony, 2024

```
clientApp.post('/readStatisticsData', authenticateJWT, async (req, res) => {
  const researcherObj = new User(req.body)
  if (researcherObj.org === "Org1") {
    console.log(researcherObj)
    try {
      console.log("FOUND PATH: ", walletPath_org1)
      responseOrg1 = await app.readStatisticsData(researcherObj, walletPath_org1);
    } catch (error) {
      console.error(`Failed to read statistics data from Org1: ${error}`);
      responseOrg1 = { error: `Failed to read statistics data from Org1` };
    }
  }

  if (researcherObj.org === "Org2") {
    try {
      responseOrg2 = await app.readStatisticsData(researcherObj, walletPath_org2);
    } catch (error) {
      console.error(`Failed to read statistics data from Org2: ${error}`);
      responseOrg2 = { error: `Failed to read statistics data from Org2` };
    }
  }

  if (researcherObj.org === "Org3") {
    try {
      responseOrg3 = await app.readStatisticsData(researcherObj, walletPath_org3);
    } catch (error) {
      console.error(`Failed to read statistics data from Org3: ${error}`);
      responseOrg3 = { error: `Failed to read statistics data from Org3` };
    }
  }
});
```

Figure 23: API for researchers reading statistical data

Blockchain and IPFS for data exchange

Before a file is uploaded on the IPFS, is encrypted with the AES algorithm. After the successful upload a unique CID associated with this file is returned. The encryption key and the CID are then stored in the blockchain so if a user wants to retrieve the file, must be authorized. An unauthorized user that manages to find the CID, can only download a non-readable file (Figure 24). The AES encryption adds an extra layer of the security, to ensure that personal medical information is restricted and safe.

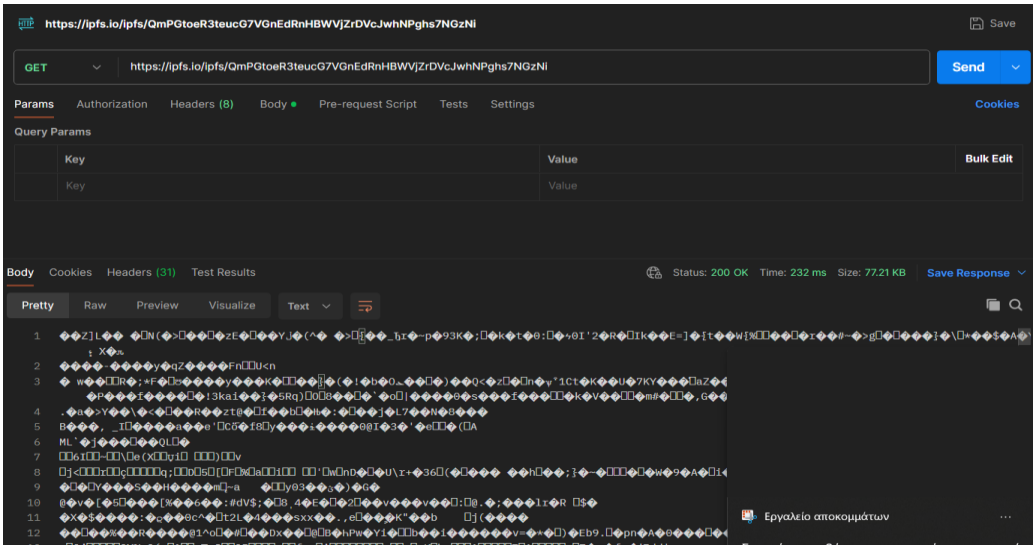


Figure 24: File is encrypted even after knowing the CID

By doing a GET request through postman to the IPFS, the requested file is encrypted and its content is not readable. However, an authorized doctor can download the file, decrypt it with the key that was stored in the blockchain and read it as it is illustrated in Figure 25.

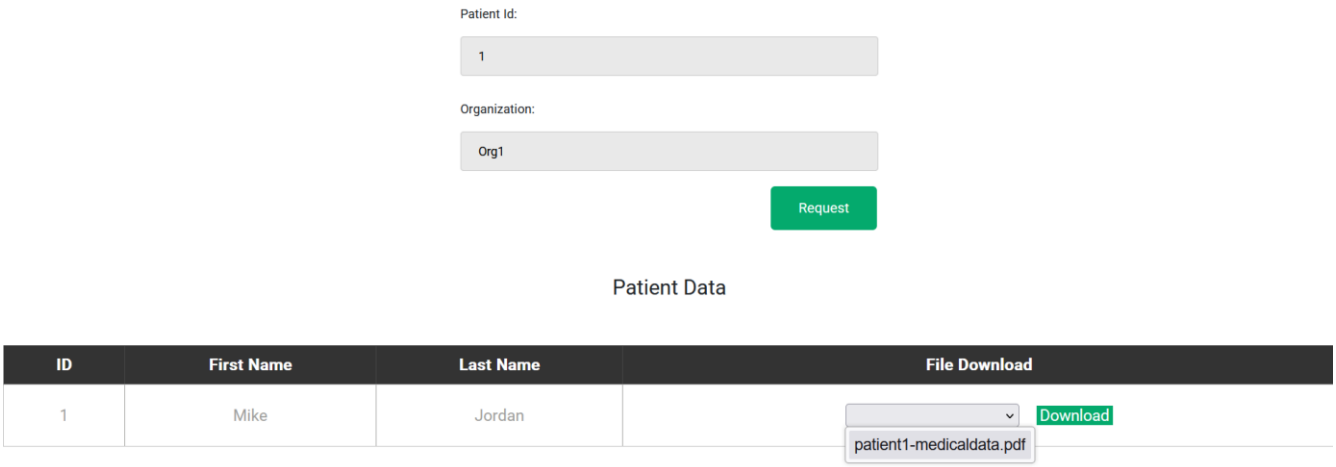


Figure 25 An authorized doctor can download patients' files

Notification System and Emergency Feature

In the present design the zero-knowledge component resides entirely in the HTTP handler that fronts the database layer, not inside the emergencyAccessPatientRecord() helper (Figure 26). When a clinician invokes POST /emergencyAccessPatientRecord (Figure 27), the request body must already contain the Groth16 proof and its accompanying public signals, both generated off-chain by the Circom circuit that guards emergency access. At the very start of the route the server loads EmergencyAccess_verification_key.json, the succinct verification key produced during the circuit's trusted setup, and retrieved by the IPFS. This key is indispensable for proof checking and is therefore read before any further processing occurs.

Immediately after loading the key, the endpoint constructs a User object from the incoming payload and aborts if the proof or public signals are missing. Proof verification is then executed with snarkjs.groth16.verify, which determines whether the submitted artefacts are both syntactically valid and semantically consistent with the arithmetic constraints of the circuit. Only if this call returns true does the execution proceed; otherwise, the request is rejected with an HTTP 400 error and no persistent state—neither in the SQL store nor on chain—is ever touched. Consequently, all cryptographic assurances are satisfied before any patient data are read or modified.

Once the proof has been accepted, control passes to emergencyAccessPatientRecord(). This function now operates under the assumption that authorisation is already cryptographically proven. Its sole responsibility is operational: it validates the doctor and patient identifiers, updates or inserts the corresponding row in hospitalInfo.patientDoctorAccess, and returns a confirmation message. Although the snippet you provided omits it, deployments typically append a follow-up call here to log the same metadata—clinician ID, patient ID, timestamp, and reason—to a blockchain ledger, thereby guaranteeing an immutable audit trail.

© CYberSynchrony, 2024

```
async function emergencyAccessPatientRecord(userObj) {
    let connection;
    try {
        connection = await pool.promise().getConnection();
        const reason = userObj.reason;
        const doctorId = parseInt(userObj.username.replace(/^doctor(\d+)$/i, '$1'));
        const patientIdNum = parseInt(userObj.patientId);
        if (!doctorId || !patientIdNum) {
            throw new Error('Invalid doctorId or patientId');
        }
        const [checkResult] = await connection.query(
            'SELECT 1 FROM hospitalInfo.patientDoctorAccess WHERE patientId = ? AND doctorId = ? LIMIT 1',
            [patientIdNum, doctorId]
        );
        if (checkResult.length > 0) {
            await connection.query(
                'UPDATE hospitalInfo.patientDoctorAccess SET doctorAccess = 1, reason = ? WHERE patientId = ? AND doctorId = ?',
                [reason, patientIdNum, doctorId]
            );
        } else {
            await connection.query(
                'INSERT INTO hospitalInfo.patientDoctorAccess (patientId, doctorId, doctorAccess, reason) VALUES (?, ?, 1, ?)',
                [patientIdNum, doctorId, reason]
            );
        }
        return { message: 'Emergency access granted successfully.' };
    } catch (error) {
        console.error(`❌ emergencyAccessPatientRecord() --> Failed to perform emergency access: ${error.message}`);
        return { error: error.message };
    } finally {
        if (connection) connection.release();
    }
}
```

Figure 26: Emergency access method

Separating proof verification from data mutation in this manner yields several advantages. First, it concentrates all zero-knowledge logic in a single, easily audited location, simplifying both maintenance and formal review. Second, it prevents any accidental leakage of sensitive information by ensuring that data-base operations occur only after cryptographic authorisation succeeds. Finally, the database helper remains narrowly scoped and testable, while the API layer continues to enforce end-to-end adherence to zero-knowledge security principles.

In parallel, the backend functionality securely logs the emergency access request on the blockchain ledger. This blockchain integration creates an immutable, tamper-proof record of the action, including key metadata such as timestamps, the requesting healthcare professional's details, and the justification for the access.

The markNotificationsAsRead() function (Figure 28) is an essential component of the notification management system, designed to help doctors efficiently track and manage their notifications. This function ensures that once a doctor has viewed a notification, it is marked as read, thereby preventing redundant notifications from cluttering the interface. By distinguishing between read and unread notifications, the system streamlines the workflow, allowing focus on the most recent and relevant updates.

© CYberSynchrony, 2024

```
clientApp.post('/emergencyAccessPatientRecord', authenticateJWT, async (req, res) => {
  try {
      let verificationKey;
      try {
        verificationKey = JSON.parse(fs.readFileSync('EmergencyAccess_verification_key.json', 'utf8'));
        console.log('Verification key loaded successfully.');
      } catch (err) {
        console.error('Error loading verification key:', err);
        process.exit(1);
      }
      const doctorObj = new User(req.body);
      console.log(doctorObj);

      if (!doctorObj.proof || !doctorObj.publicSignals) {
        return res.status(400).json({ error: 'Proof and publicSignals are required.' });
      }

      const valid = await snarkjs.groth16.verify(verificationKey, doctorObj.publicSignals, doctorObj.proof);
      if (!valid) {
        console.log('Proof verification failed for patient:', doctorObj.patientId);
        return res.status(400).json({ error: 'Invalid emergency proof provided.' });
      }
      const result = app.emergencyAccessPatientRecord(doctorObj);
      console.log('Proof verified successfully for patient:', doctorObj.patientId);
      res.json(result);
  } catch (error) {
      res.status(500).json({ error: error.message });
  }
});
```

Figure 27 API for emergency access

```
async function markNotificationsAsRead(doctorObj) {
    let connection;
    try {
        connection = await new Promise((resolve, reject) => {
            pool.getConnection((err, conn) => {
                if (err) {
                    reject(err);
                } else {
                    resolve(conn);
                }
            });
        });

        const doctorId = parseInt(doctorObj.doctorId || doctorObj.username.replace(/^doctor(\d+)$/i, '$1'));

        // Query to retrieve notifications for the doctor
        const notificationQuery = 'UPDATE notifications SET status = 1 WHERE doctorId = ? AND status = 0';
        const notifications = await queryAsync(connection, notificationQuery, [doctorId]);

        // Return the list of notifications
        return { notifications };

    } catch (error) {
        console.error(`getNotification() --> Failed to retrieve notifications: ${error}`);
        return { error: `Failed to retrieve notifications: ${error}` };
    } finally {
        if (connection) {
            connection.release();
        }
    }
}
```

Figure 28: NodeJS function implementation for marking notifications as read.

© CYberSynchrony, 2024

When the markNotificationsAsRead() function is invoked, it first establishes a secure connection to the database. The function then retrieves the doctor's unique ID from the doctorObj, ensuring that only notifications specific to that doctor are updated. It performs an SQL update operation, setting the status of all unread notifications (status = 0) to 1, which marks them as read. This update process is crucial for maintaining an organized and relevant notification feed, enhancing usability and reducing information overload.

After completing the update, the function returns an updated list of notifications to the application, reflecting the current status of each item. The function includes robust error handling to capture and address any issues encountered during the database operations, ensuring that potential failures are managed gracefully and do not disrupt the user experience. Logging mechanisms may also be employed to document any errors, supporting system maintenance and troubleshooting. The function employs efficient connection management by releasing the database connection once the operation is complete. This prevents resource bottlenecks and ensures that the database remains accessible for other users.

The getPatientNotification() function (Figure 29) enables patients to manage access requests from healthcare professionals. This function allows patients to review and respond to access requests in a secure and efficient manner. When a healthcare professional submits an access request, the system generates a notification for the patient, which is retrieved by the getPatientNotification() function. This function queries the database for any pending notifications associated with the patient, filtering results to ensure privacy and accuracy.

```
async function getPatientNotification(patientObj) {
    let connection;
    try {
        connection = await new Promise((resolve, reject) => {
            pool.getConnection((err, conn) => {
                if (err) {
                    reject(err);
                } else {
                    resolve(conn);
                }
            });
        });

        const patientId = parseInt(patientObj.patientId || patientObj.username.replace(/^patient(\d+)$/i, '$1'));

        // Query to retrieve notifications for the doctor
        const notificationQuery = 'SELECT id, message FROM requests WHERE patientId = ?';
        const notifications = await queryAsync(connection, notificationQuery, [patientId]);

        // Return the list of notifications
        return { notifications };

    } catch (error) {
        console.error(`getNotification() --> Failed to retrieve notifications: ${error}`);
        return { error: `Failed to retrieve notifications: ${error}` };
    } finally {
        if (connection) {
            connection.release();
        }
    }
}
```

Figure 29: NodeJS function implementation for retrieving patients' notifications.

The interface displays the retrieved notifications, providing details such as the identity of the requesting healthcare professional and the specific nature of the request. Patients can then make informed decisions about whether to grant or deny access to their records. Once the patient takes action, the status of the request is updated via the changeStatus() function on the backend, and this action is securely logged on the blockchain.

In parallel, the emergency workflow is supported by the emergencyAccessPatientRecord() function, which addresses critical scenarios where immediate access to patient data is required. This function is invoked by

© CYberSynchrony, 2024

healthcare professionals during emergencies and is designed to provide secure, time-limited access to patient records without prior consent. The backend processes the request by first validating the identities of the patient and the requesting professional and then either updating an existing access record or creating a new one. This function also logs the action on the blockchain, ensuring a transparent and immutable record of the emergency access request.

Upon execution, the getPatientNotification() function initiates a connection to the database and extracts the patient ID from the patientObj, ensuring that the notification retrieval process is specific to the individual patient. It then queries the requests table to fetch relevant notifications, filtering based on the patient ID to guarantee that only notifications concerning the patient's data access requests are retrieved. This ensures privacy by isolating notifications that pertain exclusively to the individual.

The function provides patients with detailed notifications, including information on which doctor is requesting access and, potentially, the reason for the request. Patients can review each notification and respond accordingly. The function incorporates comprehensive error handling to capture and log any potential issues during the retrieval process. For instance, an error might occur if the database connection times out due to high traffic or network instability. In such cases, the function logs the error details, notifies the appropriate system administrator, and displays a user-friendly message. Once notifications are successfully retrieved or any error is handled, the function releases the database connection to promote efficient resource utilization and prevent bottlenecks in the system.

These functions lay the foundation for a secure and transparent communication system between patients and healthcare providers. By recording interactions on both the database and blockchain, the system ensures reliable and accountable management of healthcare data access.

Security and Privacy

As was mentioned in section 2, many approaches have been followed to ensure secure functionality of the blockchain architecture in a multilayer context. First, for the authentication process, JSON Web Tokens (JWTs) have been used to securely transmit information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. By using JWTs efficiently, the identity of users can be validated and information about user permissions can be transmitted in a compact way through the JSON objects, making them a popular choice in modern web application development for managing user sessions and API authentication. The following UML diagram (Figure 30) represents the steps for the user authentication process using JWTs.

© CYberSynchrony, 2024

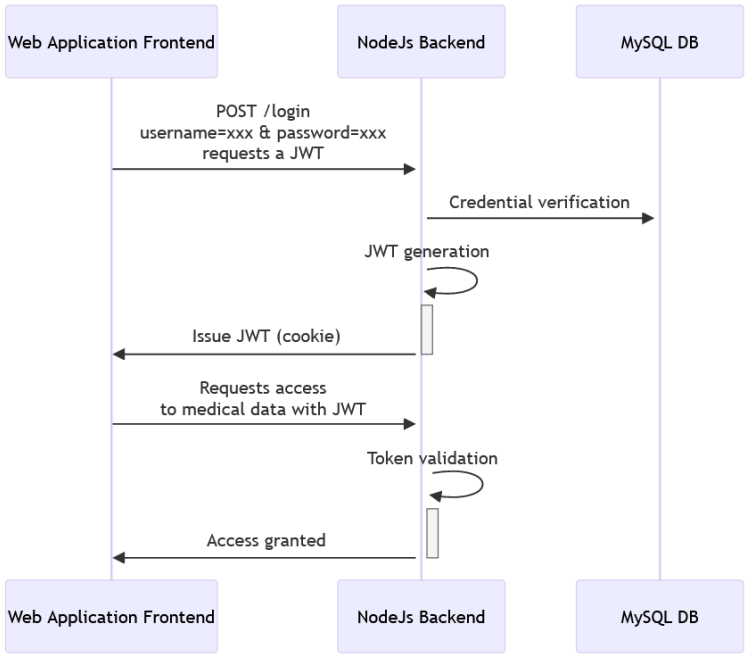


Figure 30: UML diagram for user authentication.

The user first tries to log in through the web application by sending a POST request containing credentials. After the credentials are verified, a unique JWT is generated and sent back to the user inside a cookie. The user can then request access to medical information using that JWT. After validating the token, the user is able to retrieve the data.

File Security

IPFS provides strong security and data integrity through decentralized storage. Unlike conventional centralized systems, which are vulnerable to single points of failure, IPFS allows data to be shared globally over a distributed network of nodes. This reduces the risks typically associated with centralized storage, such as data breaches or permanent loss.

Each piece of data on IPFS has a unique identity represented by a cryptographic hash, known as the Content Identifier (CID). Any change in the data will result in a different CID. This ensures immutability and the integrity of stored medical records.

However, a key challenge is that if an unauthorized user obtains the CID, they can retrieve the file from the IPFS network. To mitigate this, files are encrypted using the AES (Advanced Encryption Standard) algorithm before being uploaded. Even if an unauthorized user gains access to the CID and downloads the file, they cannot decrypt the contents without the corresponding decryption key.

The efficiency of AES encryption was tested by uploading files of various sizes (10, 20, 30, 40, and 50 MB). Figure 31 shows the execution times for encryption:

© CYberSynchrony, 2024

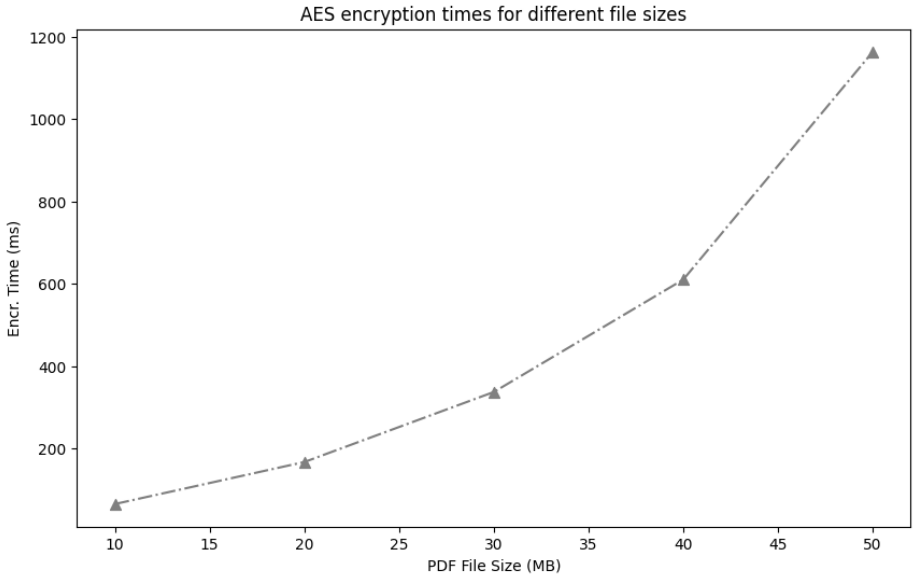


Figure 31: AES execution time for encrypting files of various sizes.

In Figure 32, the total execution time (encryption + IPFS upload) is illustrated. As expected, upload times vary and are not directly correlated with file size due to factors such as network congestion. For example, uploading a 50 MB file took approximately 6057 ms, while a 40 MB file took 6467 ms—slightly longer despite the smaller size (see Table 5).

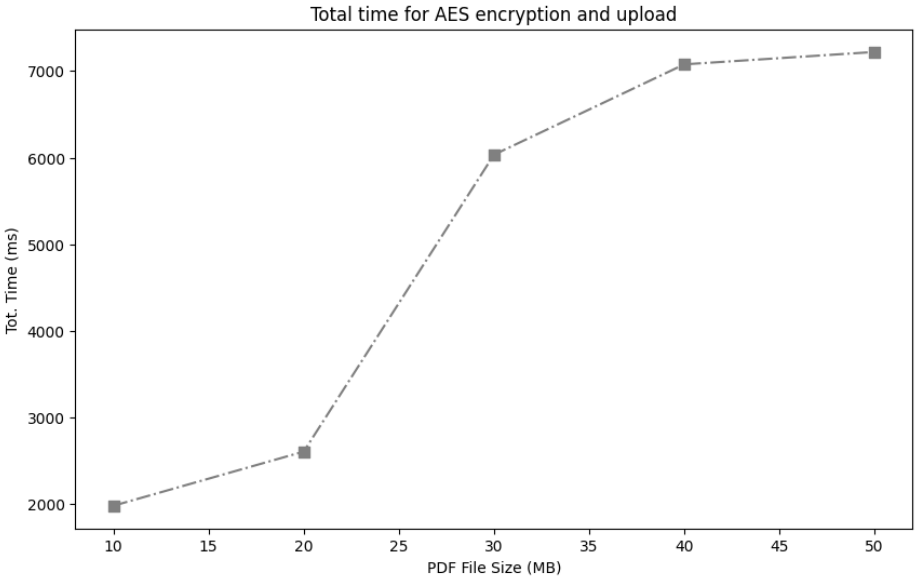


Figure 32: Total time for file encryption and IPFS upload.

To ensure security post-upload, both encryption keys and CIDs are stored immutably on the blockchain. Smart contracts are deployed to define who, when, and under what conditions the CID can be accessed. These contracts enforce access policies set by the patient or authorized healthcare providers, ensuring strict control over sensitive medical records.

Table 5: Summarized results for encryption and uploading times.

File Size (MB)	***Encryption Time (ms)***	***Total Time (ms)***
10	65	1981

© CYberSynchrony, 2024

20	167	2606
30	337	6034
40	610	7077
50	1163	7220

Our solution has implemented a PoC demonstration (Figure 33) integrating Differential Privacy (DP), Homomorphic Encryption (HE), and Blockchain. This way, we enable secure and privacy-preserving data analysis across distributed data sources for researchers. This architecture ensures that sensitive data can be analyzed collaboratively without compromising individual privacy or utility.

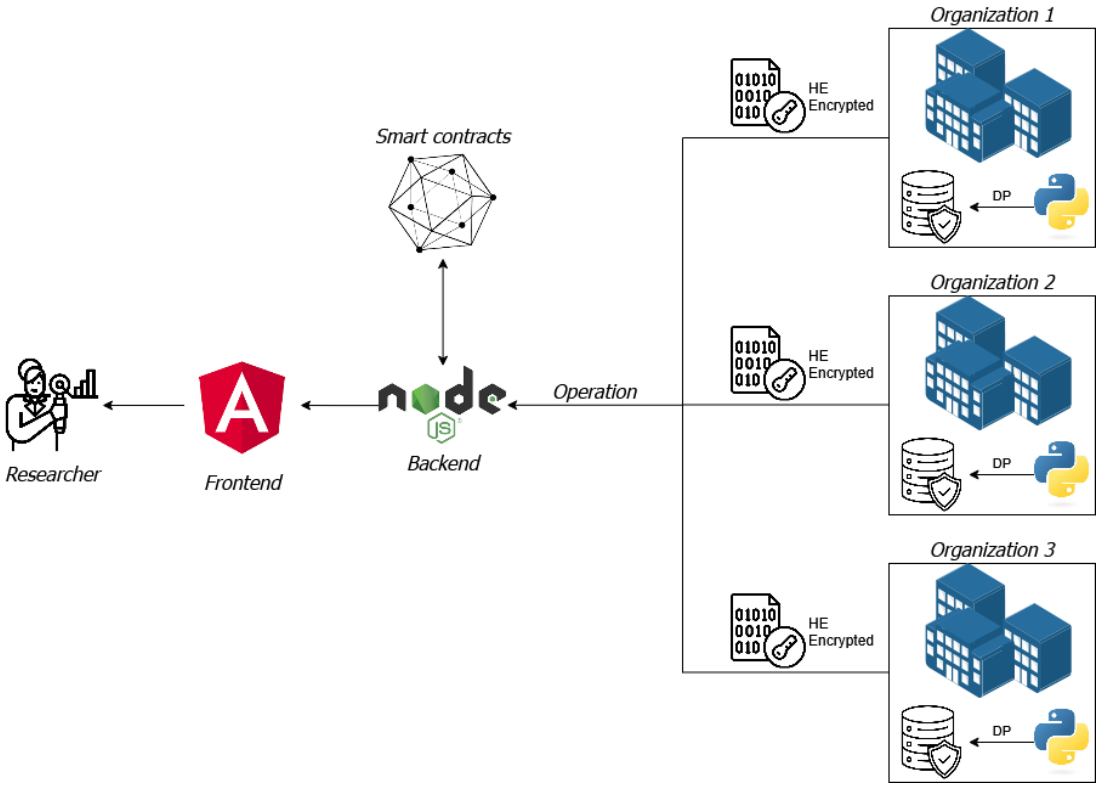


Figure 33: Unified Framework for Differential Privacy, Homomorphic Encryption, and Blockchain in Privacy-Preserving Data Analysis

As shown in the diagram, each participating entity begins by applying Local Differential Privacy to anonymize data at the individual record level before any computations take place. This step ensures that privacy is preserved locally and that no raw data is transmitted externally.

When external organizations or third-party data providers contribute datasets, Homomorphic Encryption is used to encrypt data before transmission. This allows encrypted data to be securely integrated into the analysis pipeline and ensures that computations can be performed without decryption. After analysis is complete, only authorized users are permitted to decrypt the final aggregated results.

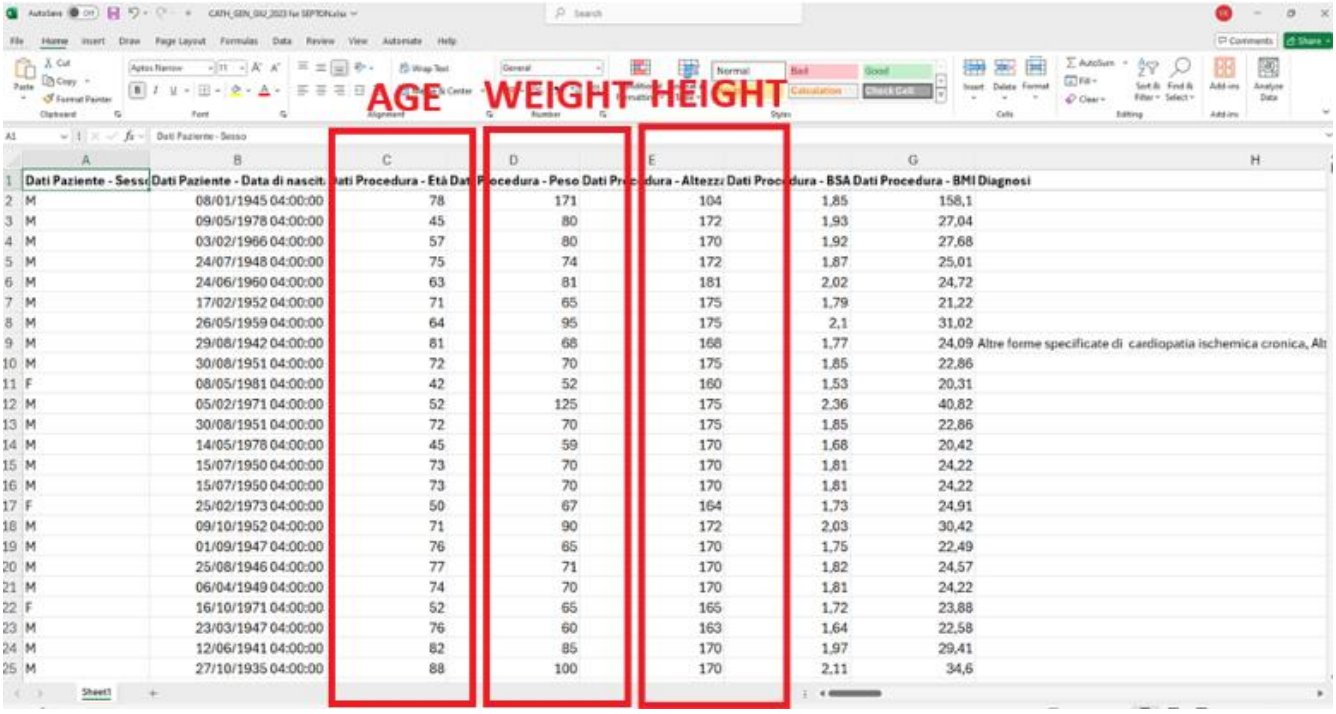
This layered privacy-preserving architecture combines the strengths of Differential Privacy, Homomorphic Encryption, and Blockchain to deliver a robust, scalable, and secure solution for collaborative data analysis. It supports secure data sharing, real-time decision-making, and compliance with data protection requirements.

As an example, we might think of a user who can issue a query such as: “How many individuals aged over 60 are classified under a certain condition across distributed data sources?” This triggers the privacy-preserving pipeline, which uses the integrated framework to compute the requested statistics securely and collaboratively.

© CYberSynchrony, 2024

Throughout the process, no raw data is revealed, and privacy is maintained at each step. The result (an aggregated value) is then returned to the user without exposing any sensitive information.

This PoC demonstrator has been implemented and tested using a **synthetic dataset** designed to simulate the structure and format of real-world records (Figure 34). This dataset included fields such as age, weight, height, and categorical attributes. As the dataset is fully synthetic, it enabled realistic validation of the framework without exposing any real or sensitive information.



	A	B	C	D	E	G	H
1	Dati Paziente - Sess	Dati Paziente - Data di nascit	ati Procedura - Età Dat	Procedura - Peso Dati Pro	dura - Altezz Dati Proc	dura - BSA Dati Procedura - BMI Diagnosi	
2	M	08/01/1945 04:00:00	78	171	104	1,85 158,1	
3	M	09/05/1978 04:00:00	45	80	172	1,93 27,04	
4	M	03/02/1966 04:00:00	57	80	170	1,92 27,68	
5	M	24/07/1948 04:00:00	75	74	172	1,87 25,01	
6	M	24/06/1960 04:00:00	63	81	181	2,02 24,72	
7	M	17/02/1952 04:00:00	71	65	175	1,79 21,22	
8	M	26/05/1959 04:00:00	64	95	175	2,1 31,02	
9	M	29/08/1942 04:00:00	81	68	168	1,77 24,09 Altre forme specificate di cardiopatia ischemica cronica, Alt	
10	M	30/08/1951 04:00:00	72	70	175	1,85 22,86	
11	F	08/05/1981 04:00:00	42	52	160	1,53 20,31	
12	M	05/02/1971 04:00:00	52	125	175	2,36 40,82	
13	M	30/08/1951 04:00:00	72	70	175	1,85 22,86	
14	M	14/05/1978 04:00:00	45	59	170	1,68 20,42	
15	M	15/07/1950 04:00:00	73	70	170	1,81 24,22	
16	M	15/07/1950 04:00:00	73	70	170	1,81 24,22	
17	F	25/02/1973 04:00:00	50	67	164	1,73 24,91	
18	M	09/10/1952 04:00:00	71	90	172	2,03 30,42	
19	M	01/09/1947 04:00:00	76	65	170	1,75 22,49	
20	M	25/08/1946 04:00:00	77	71	170	1,82 24,57	
21	M	06/04/1949 04:00:00	74	70	170	1,81 24,22	
22	F	16/10/1971 04:00:00	52	65	165	1,72 23,88	
23	M	23/03/1947 04:00:00	76	60	163	1,64 22,58	
24	M	12/06/1941 04:00:00	82	85	170	1,97 29,41	
25	M	27/10/1935 04:00:00	88	100	170	2,11 34,6	

Figure 34: Synthetic Dataset Example for Privacy-Preserving Healthcare Data Analysis

The following Python script (Figure 35) demonstrates the PoC of differential privacy using the PyDP library to compute private statistics (specifically bounded mean, bounded sum, and count) on the previously discussed dataset containing age, weight, and height information. In the first stage, a Flask service (port 5000) uses the **PyDP** Laplacian mechanisms *BoundedMean*, *BoundedSum* and *Count* to publish differentially-private statistics for three "virtual" hospitals. The dataset is split into equal thirds, and for each slice the service returns the real mean/sum as well as the private counterpart computed with ε = 0.01 and tight lower/upper bounds (Age 21-94, Weight 40-180 kg, Height 46-200 cm). A smaller epsilon gives strong privacy at the cost of additional noise; bounding every column keeps the global sensitivity low and therefore preserves more utility. The endpoint /hospital_stats/<hospital_id> emits a JSON payload containing real and DP values side-by-side, so an analyst can see exactly how the privacy budget distorts each statistic.

© CYberSynchrony, 2024

```python
def calculate_statistics(hospital_df):
    stats = {}

    def bounded_mean(column, epsilon, delta, lower_bound, upper_bound):
        bounded_mean = BoundedMean(epsilon=epsilon, delta=delta, lower_bound=lower_bound, upper_bound=upper_bound)
        return float(bounded_mean.quick_result(column.tolist()))

    def bounded_sum(column, epsilon, delta, lower_bound, upper_bound):
        bounded_sum = BoundedSum(epsilon=epsilon, delta=delta, lower_bound=lower_bound, upper_bound=upper_bound)
        return float(bounded_sum.quick_result(column.tolist()))

    def count(column, epsilon):
        dp_count = Count(epsilon=epsilon)
        return int(dp_count.quick_result(column.tolist()))

    stats['age_mean_real'] = float(hospital_df["Age"].mean())
    stats['age_mean_dp'] = bounded_mean(hospital_df["Age"], epsilon=0.01, delta=0.0, lower_bound=21, upper_bound=94)
    print('--------------------Real Age Mean: '+str(stats['age_mean_real'])+' | DP Age Mean :'+str(stats['age_mean_dp'])+'------------------------')
    stats['weight_mean_real'] = float(hospital_df["Weight"].mean())
    stats['weight_mean_dp'] = bounded_mean(hospital_df["Weight"], epsilon=0.01, delta=0.0, lower_bound=40, upper_bound=180)
    print('--------------------Real Weight Mean: '+str(stats['weight_mean_real'])+' | DP Weight Mean :'+str(stats['weight_mean_dp'])+'------------------------')
    stats['height_mean_real'] = float(hospital_df["Height"].mean())
    stats['height_mean_dp'] = bounded_mean(hospital_df["Height"], epsilon=0.01, delta=0.0, lower_bound=46, upper_bound=200)
    print('--------------------Real Height Mean: '+str(stats['height_mean_real'])+' | DP Height Mean :'+str(stats['height_mean_dp'])+'------------------------')
    stats['age_sum_real'] = float(hospital_df["Age"].sum())
    stats['age_sum_dp'] = bounded_sum(hospital_df["Age"], epsilon=0.01, delta=0.0, lower_bound=21, upper_bound=94)
    print('--------------------Real Age Sum: '+str(stats['age_sum_real'])+' | DP Age Sum :'+str(stats['age_sum_dp'])+'------------------------')
    stats['weight_sum_real'] = float(hospital_df["Weight"].sum())
    stats['weight_sum_dp'] = bounded_sum(hospital_df["Weight"], epsilon=0.01, delta=0.0, lower_bound=40, upper_bound=180)
    print('--------------------Real Weight Sum: '+str(stats['weight_sum_real'])+' | DP Weight Sum :'+str(stats['weight_sum_dp'])+'------------------------')
    stats['height_sum_real'] = float(hospital_df["Height"].sum())
    stats['height_sum_dp'] = bounded_sum(hospital_df["Height"], epsilon=0.01, delta=0.0, lower_bound=46, upper_bound=200)
    print('--------------------Real Height Sum: '+str(stats['height_sum_real'])+' | DP Height Sum :'+str(stats['height_sum_dp'])+'------------------------')

    stats['count_real'] = int(len(hospital_df))
    stats['count_dp'] = count(hospital_df["Age"], epsilon=0.01)

    return stats

@app.route('/hospital_stats/<int:hospital_id>', methods=['GET'])
def get_hospital_statistics(hospital_id):
    hospitals = load_and_split_data()
    if 1 <= hospital_id <= 3:
        hospital_df = hospitals[hospital_id - 1]
        stats = calculate_statistics(hospital_df)
        return jsonify(stats)
    else:
        return jsonify({"error": "Invalid hospital ID"}), 400

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Figure 35 Differentially private computation of mean, sum, and count using PyDP, protecting individual values while enabling aggregate insights.

The second stage—served from a separate Flask app on port 5007—fetches those DP **sums** from all three hospitals and protects them further with additive homomorphic encryption. Using **Pyfhel** (BFV scheme, n = 2^{14}, t = 20 bits), the script encrypts each hospital's *age-sum*, *weight-sum*, and *height-sum*, then adds the ciphertexts together in encrypted space. No raw data—or even the noisy partial sums—are exposed during that aggregation. Only after the totals have been computed does the script decrypt, revealing the global DP sums. For demonstration purposes it also serialises each final ciphertext and returns the last 20 bytes in hex.

After the encrypted computations are completed, the final results are passed to the frontend through a Flask API, allowing researchers to securely access the outcomes via a web interface. This setup ensures that authorized researchers can retrieve insights without ever exposing raw or sensitive data during transmission or interaction.

© CYberSynchrony, 2024

```python
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/encrypt_sums', methods=['GET'])
def encrypt_sums():
    HE = Pyfhel()
    HE.contextGen(scheme='bfv', n=2**14, t_bits=20)
    HE.keyGen()
    zero_np = np.array([0], dtype=np.int64)
    enc_age_sum = HE.encryptInt(zero_np)
    enc_weight_sum = HE.encryptInt(zero_np)
    enc_height_sum = HE.encryptInt(zero_np)
    for hospital_id in [1, 2, 3]:
        url = f"http://localhost:5000/hospital_stats/{hospital_id}"
        response = requests.get(url)
        data = response.json()
        dp_age_sum = int(round(data['age_sum_dp']))
        dp_weight_sum = int(round(data['weight_sum_dp']))
        dp_height_sum = int(round(data['height_sum_dp']))
        enc_age_hosp = HE.encryptInt(np.array([dp_age_sum], dtype=np.int64))
        enc_weight_hosp = HE.encryptInt(np.array([dp_weight_sum], dtype=np.int64))
        enc_height_hosp = HE.encryptInt(np.array([dp_height_sum], dtype=np.int64))
        enc_age_sum += enc_age_hosp
        enc_weight_sum += enc_weight_hosp
        enc_height_sum += enc_height_hosp
    dec_age_sum = HE.decryptInt(enc_age_sum)[0]
    dec_weight_sum = HE.decryptInt(enc_weight_sum)[0]
    dec_height_sum = HE.decryptInt(enc_height_sum)[0]
    enc_age_bytes = enc_age_sum.to_bytes()
    enc_weight_bytes = enc_weight_sum.to_bytes()
    enc_height_bytes = enc_height_sum.to_bytes()
    enc_age_hex = enc_age_bytes[len(enc_age_bytes)-20:].hex()
    enc_weight_hex = enc_weight_bytes[len(enc_weight_bytes)-20:].hex()
    enc_height_hex = enc_height_bytes[len(enc_height_bytes)-20:].hex()

    result = {
        "Encrypted Age Sum (last 20 bytes)": enc_age_hex,
        "Encrypted Weight Sum (last 20 bytes)": enc_weight_hex,
        "Encrypted Height Sum (last 20 bytes)": enc_height_hex,
        "Decrypted Age Sum": abs(int(dec_age_sum)),
        "Decrypted Weight Sum": abs(int(dec_weight_sum)),
        "Decrypted Height Sum": abs(int(dec_height_sum)),
    }
    return jsonify(result)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5007, debug=True)
```

Figure 36 Homomorphic encryption using Pyfhel library

© CYberSynchrony, 2024