FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Identity Management Solution: Security and Trust through Blockchain Technology

Jénifer Graça Gouveia Constantino

Masters in Software Engineering

Supervisor: António Miguel Pimenta Monteiro

Supervisor: Pedro Miguel Freire Berenguer

July 20, 2024

Identity Management Solution: Security and Trust through Blockchain Technology

Jénifer Graça Gouveia Constantino

Masters in Software Engineering

July 20, 2024

Abstract

Through the years, **document forgery** and **counterfeiting** have increasingly been felt in different sectors, such as education and financial institutions.

Many techniques have been used to prevent this falsification, including Barcode, Hashing, Document Signatures, Watermarks, Human Analysis, AI, and Document-Sealing Solutions. Nevertheless, these are either **not efficient** or **centralized**.

The current dissertation **doesn't aim** to solve an unbreakable problem, instead, it focuses on how the problem is being addressed. The thesis **proposes** an analysis of a decentralized document management, tracker, share, and verifier system oriented to decentralization over security. The current dissertation proposes an analysis of how security and decentralization can go hand in hand when developing a dApp, containing import documents, over a public blockchain such as Ethereum. It also explains the development process alongside good code and security practices.

First, a technology analysis was performed, followed by the development of a proof of concept, and finally, unit tests were integrated alongside test and security tools. Multiple **research questions** were defined in the literature review and answered in the conclusion. It was possible to understand that a 100% decentralized application is not possible due to the decentralization issues the blockchain itself has. By developing in a public blockchain, several challenges were posed to guarantee document integrity and confidentiality. Ethereum proved not to be user-friendly since every user's action was translated into a high-cost transaction. Overall, the developed solution proved its integrity and robustness by acquiring more than **90% of unit test coverage** and by **not having any security problems and warnings** detected by the front-end and back-end security tools.

Acknowledgement

I want to express my gratitude to Professor António Monteiro from Porto University for his invaluable support, the provision of essential materials, and his guidance through our meetings during the semester.

I also thank Pedro Berenguer from Nearsoft for the insightful meetings where he imparted extensive knowledge on blockchain and related topics.

I am deeply grateful to Professor Ana Paiva from Porto University for her guidance in conducting research and writing a comprehensive state of the art.

And lastly, I would like to thank the ACM Blockchain Association for their assistance and expertise in dApp development, especially Tiago Loureiro and Pedro Costa.

Their insights and feedback helped me write and develop the current thesis, ”Identity Management Solution: Security and Trust through Blockchain Technology”.

Jénifer Graça Gouveia Constantino

"Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly."

Vitalik Buterin

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
BGP	Border Gateway Protocol
dApp	Decentralized Application
DLT	Distributed Ledger Technology
DoS	Denial of Service Attacks
ECC	Elliptic Curve Cryptography
EVM	Ethereum Virtual Machine
FAN	Fork After Withholding
HD Wallet	Hierarchical Deterministic Wallet
ICP	Internet Computer
ICP	Internet Computer Protocol
IPFS CID	InterPlanetary File System Content Identifier
IV	Initialization Vector
MVP	Minimum Viable Product
P2P	Peer-to-Peer
PKI	Public Key Infrastructures
PoS	Proof of Stake
PoW	Proof of Work
RBAC	Role Based Access Control
XSS	Cross Site Scripting
ZK-SNARKs	Zero-knowledge Succinct Non-interactive Arguments of Knowledge

Chapter 1

Introduction

Many solutions exist to solve the **problem** of document forgery and counterfeiting. Section "Literature Review," further details the problem and the existing solutions. A **centralized** solution partially addresses the indicated problem but has significant drawbacks. It has **major security flaws** because all data resides with a single entity, making it a prime attack target. Additionally, it suffers from **severe transparency issues**, as one entity has complete control over the system. This dissertation's proposed solution leverages blockchain technology's characteristics to deliver a solution that promotes document immutability and confidentiality through a transparent and decentralized process.

To further understand what will be explained in the coming chapters, this section provides some ground knowledge to help the reader understand the document better, navigate through more complex topics, and have the necessary context for a complete understanding.

This section describes **blockchain** concepts such as gas, mining, forks, and many others. Alongside these concepts, it details blockchain security features, vulnerabilities, and problems. Following the same structure, a detailed explanation of **dApps** concepts and dApps security and vulnerability features is provided. It further details **concepts** such as encryption, hashing, hierarchical deterministic wallets (HD), authentication, authorization, and session timeout mechanisms. Finally, a brief explanation of the **dissertation structure** is provided.

1.1 Blockchain

Blockchain is a chain of blocks, in which each **block** contains a **group of transactions**. Each of these blocks is **connected** through a **hash value** calculated based on the previous block's hash.

Each block has a **maximum number of transactions**, be it because there is a **block size** pre-established (Bitcoin) or because there is a **gas limit** (Ethereum). Therefore, blockchain can be considered secure, transparent, immutable/tamper resilient, distributed, decentralized, traceable, automated, and anonymous. [118]

According to [62], in Ethereum, **gas** is a unit that measures the amount of computational power required to execute an operation, such as transactions or smart contracts. While **smart contracts** are self-executing pieces of code that translate agreements, **transactions** are fundamental operations that move data, currency, or information from one point to another within the network. The smart contract execution is performed through a transaction.

Actions that imply a modification in the blockchain imply gas consumption, and these are called **transactions**, but actions that only read from the blockchain state are called "calls", and these don't require gas consumption.

The required gas is proportional to the computational power required to operate. **Gas Price** is the amount of Ether a user is willing to pay per gas unit, typically measured in gwei, $1\text{gwei}=10^{-9}\text{ETH}$. Miners prefer transactions with higher fees, making transactions with high gas fees to be quickly executed, and transactions with low gas fees will possibly never be executed. The **total fee** is calculated by multiplying the total gas used with the gas price. Before operating, a gas limit has to be defined. **Gas Limit** is the maximum amount of gas a user is willing to pay on a transaction. If too low, the transaction will be aborted - changes will be rolled back, and all gas will be spent and lost. On the other hand, if the gas limit is higher than necessary to execute the operation, the remaining gas will be returned to the user. Nevertheless, it's not recommended to set a value too large for the gas limit; if any bug occurs during the transaction execution, all gas will be lost. **Factors affecting** the gas consumption are:

- Type of operation. Complex smart contracts consume more gas than simple operations.
- Network Congestion. To ensure transactions are executed quicker, transaction executors have to set up a better gas price for them to be chosen by the miners.
- Code Efficiency. Efficient smart contracts consume less gas.

As such, gas is used to calculate fees paid by the transaction executor and received by the miner as compensation for the computation resources they provide. [62]

Blockchains, such as Bitcoin and Ethereum, are secured by a **mining process**, which consists of adding new blocks to the blockchain. **Miners**, the nodes responsible for mining, group transactions into a block, perform a consensus mechanism, and broadcast the mined block into

the network. When a block is added to the main chain, miners receive a block reward and a transaction inclusion fee, more detailed in [121]. [35] [118]

For nodes to agree on the legitimacy of transactions and the current state of the blockchain, a **consensus mechanism** is used. Among them, the most popular are **PoW**, used by Bitcoin, and **PoS**, used by Ethereum. [118] **PoW** focuses on the miners' computational power; they all work simultaneously to find the solution for a problem, which is solved by incrementing a nonce several times until the current block hash reaches a certain number of leading zeros. The miner who discovers the nonce first secures the block and earns the mining reward, while the efforts of the other miners go unrewarded. On the other hand, **PoS** focuses on the miner's wealth. Each node should have a certain amount of value staked, and the higher the stake, and the time in which that value is stacked, the higher the probability of the miner being chosen to mine the block. If the user tries to deceive the network, the stake is taken out from the user. This consensus mechanism is less energy dependent because only one miner is mining, and there is no need for a nonce. Each consensus mechanism influences the blockchain processing ability, scalability, and security in different ways [128]. More information can also be found in articles [37] and [35].

Because mining is getting more costly and competitive, especially in PoW blockchains, miners group themselves into **mining pools**. Here, miners share computational power, increasing the probability of receiving a block reward. Once this is achieved, the value of the reward is distributed among the participating miners. [118]

During mining, **forks** might happen. There can be accidental forks or intentional forks. Because PoW allows parallel mining, blocks might reference to the same predecessor, originating **Accidental Forks**. This type of fork can be solved using the '**longest chain rule**,' used by the Bitcoin blockchain, or using the '**Greedy Heaviest Observed Sub-tree (GHOST) protocol**,' used by the Ethereum blockchain. These are better detailed in [118] [121]. Furthermore, because updates in the blockchain might happen, **Intentional Forks** are also possible. The figures below illustrate both of these protocols.

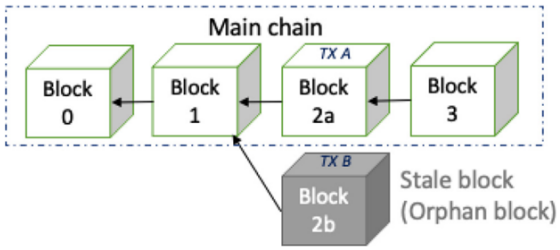


Figure 1.1: Bitcoin chooses the main chain based on the longest chain rule. Blocks outside the main chain are called stale blocks [118].

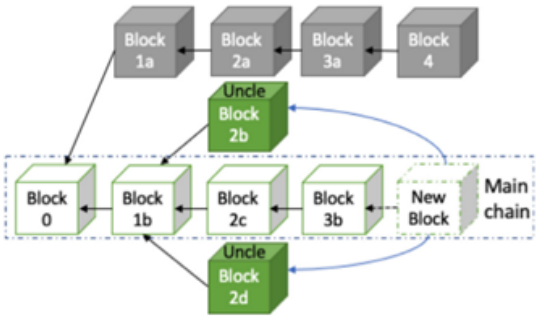


Figure 1.2: Ethereum chooses the main chain based on the GHOST protocol [118].

Lastly, these types of forks can lead to hard or soft forks. In **hard forks**, nodes with the most recent protocol version can not interoperate with nodes with the old version, which can lead to a new cryptocurrency. After the fork, the cryptocurrency owners would have the same amount in the new cryptocurrency. Afterward, these cryptocurrencies assume different values according to how the market adheres to each. Note that this doesn't mean the user has more coins. The user starts having the same amount represented by different coins. Further from this, hard forks may also happen because of philosophical reasons; this is because the community of the blockchain has deep, fundamental disagreements about the project's vision. On a **soft fork**, there is support for compatibility with nodes running on the old version of the protocol, but new transactions created by the new protocol won't be accepted by nodes running on the old version. [122]

Blockchain can be classified based on the nature of data accessibility, which defines who can and cannot access the network. And based on the need for authorization, which defines how nodes in a network can participate and what actions they can perform.

Based on its **nature of data accessibility**, blockchain can be considered **public**, anyone can join and participate in the network [7], **private**, one entity defines who is allowed to participate

in the network, **community/consortium**, where a consortium of organizations, in a closed and controlled environment, manage who joins the network [64], and **hybrid**, which combines the three previously mentioned blockchains. [100]

Based on the **need for authorization**, blockchain can be considered **permissionless**, anyone has the same roles in the network, **permissioned**, where there are different levels of user permissions/roles for different nodes, and **Hybrid**, which allows support to permissioned and permissionless blockchains.[34]

The permissions over a blockchain network are not performed at the smart contract level, where logic is deployed on the chain, but are defined at the network level. At this level, rules, protocols, and node interactions are defined.

It's possible to have a **public permissionless** blockchain but still implement roles at the contract level. Implementing **public permissioned** blockchains is also possible, where every node can join the network. Still, different nodes have different privileges defined at the network level based on how nodes interact between themselves and how consensus is achieved. [75]

Furthermore, it's possible to have **private permissionless** blockchains, where a node can be added to the chain based on specific criteria; other nodes will acknowledge its presence but won't transfer any data, unlike what happened in public blockchains. In these types of blockchains, nodes don't operate over a single chain; instead, each smart contract has its own ad-hoc chain. A node can request a copy of a private ad-hoc chain and start participating in the contract. [75]

While **private permissioned** blockchains assume advantages for regulatory reasons, **public and permissionless** blockchains enhance decentralization.[129]

Based on its **core functionality and smart contract support**, a blockchain can be **stateless**, focusing on transaction optimization and chain functionality, not providing smart contract implementation, and **stateful blockchain**, providing smart contracts and transaction computing. [100]

1.1.1 Blockchain and its decentralization issues

Even though blockchain stores data across multiple nodes (DLT), provides consensus mechanisms (PoW, PoS), and operates globally, it still faces decentralization issues.

- **Consensus Mechanisms**. Even though they provide a way to achieve agreement across multiple nodes in the network, depending if it is a PoW or PoS consensus, they provide vulnerabilities when it comes to decentralization. While PoW centralizes the network by

the nodes with higher computational power or by large mining pools, PoS centralizes the network due to unequal resource distribution - the more coins or tokens a person holds, the more influence they have over consensus. [59]

- **Centralized Governance**. If decisions such as protocol changes or updates are taken only by the blockchain developers or even foundation members, the decentralization can be compromised.
- **Centralized Infrastructure**. Many blockchains rely on centralized services such as APIs or wallets, which may become points of failure.
- **Sybil Attacks**. These attacks consist of creating fake identities to gain influence over the network, putting its decentralization at risk. [59]
- **Forks**. While forks are a decentralization mechanism, they can also lead to fragmentation, leading to fewer nodes in each originated blockchain, making them more susceptible to 51% attacks, for example. [122] [36]

1.1.2 Blockchain security features

It is important to consider security since the blockchain can hold sensitive data from users. Blockchain implements diverse security features to promote secure transaction execution, data integrity and guarantee authorized access. Everyone in the network is responsible for ensuring this blockchain security, from nodes that validate transactions using a consensus mechanism to developers responsible for updating and improving the blockchain security and protecting against security threats. [68] [103] These security features include:

- **Decentralization**. Because blockchain is a distributed ledger that uses a peer-to-peer network, there is no single point of failure. Each node has a copy of the blockchain, which makes it incredibly difficult for attackers to compromise the system. Consensus mechanisms, like PoW and PoS, ensure that all nodes agree upon a transaction's validity, avoiding fraud. [103]
- **Cryptography and Integrity**. Each block contains a cryptographic hash calculated based on its content and the cryptographic hash of the previous block. By changing the content of a block, its hash also changes and, consequently, the hash of the upcoming blocks. Furthermore, transactions are signed using the users' private keys and decrypted with the corresponding public key, ensuring that only authorized users initiate a transaction. [131]

- **Immutability**. Once a transaction is recorded in a block, it cannot be changed, as seen in the "Cryptography" point. Furthermore, each block has a timestamp, allowing a time order in which certain transactions were executed. These features allow tamper-proof records of transactions. [68]

- **Transparency, Traceability, and Auditability**. In both public and private blockchains, even though in different ways, transparency is enhanced. It is possible to track all transactions, enhancing traceability and enabling an easy verification of authenticity and data integrity. [103]

- **Smart Contracts**. Self-executing contracts containing agreements. Once deployed into the blockchain, they cannot be changed and are executed based on predefined rules. This removes the need for intermediaries, reduces the risk of fraud, and implies better compliance with regulations. [68]

- **Consensus Mechanisms**. Transactions are validated and added to the blockchain based on a consensus mechanism, like PoW and PoS, that prevents fraudulent activities like double-spending. They are also a way to prevent Sybil attacks since it is hard for an entity to control a network through multiple fake identities. Nevertheless, these attacks are still susceptible to happen. [103]

- **Access Control**. Permissioned blockchains allow us to define who can join the network and who can validate transactions. Advanced identity management systems can ensure that a user is authenticated and authorized before he can interact with the blockchain. [103]

- **Fault Tolerance**. Blockchains are resilient to Byzantine Fault Tolerance; they can reach consensus even if some nodes are malicious. [103]

1.1.3 Blockchain vulnerabilities

Even though blockchain has the previously mentioned security features, this young technology is still not immune to attacks. Various hacks have been successfully performed, as is the famous case of "The DAO hack". It's important to know the attack vectors that can possibly be performed on this technology to be aware of possible vulnerabilities that any solution developed over it can inherit.

For a better organization of the current document, in appendix A, it's possible to see an overview of the possible blockchain attacks. These attacks were separated into groups: 'Consensus Mechanism' attacks, 'Peer-to-Peer System' attacks, 'Application Oriented' attacks, and

'Smart Contract' attacks. Each group is represented in a table, and each attack is briefly explained and associated with an article that further details it. In this section, only some of these attacks will be further analyzed.

- **Majority 51% Attack**. It happens when a miner or a group of miners, for example, in a mining pool [103], own more than 50% of the network mining power, guaranteeing them enough power to manipulate the consensus mechanism. [87] In **PoW**, the attacker needs to have more than 50% of the network computational power, and in **PoS** the attacker needs to have more than 50% of the committed stake, this is, more than 50% of the networks wealth. [36] Because the attacker controls most of the network, he can overthrow the consensus mechanisms, reject blocks, and add malicious content to the blockchain [36]. This might result in more sophisticated attacks such as **double-spending** [87]. This attack is more frequent in public and permissionless blockchains, where no central authority controls the network, than in permissioned blockchains. [60] Furthermore, the older and the more distributed a network is, the less prone it is to this attack since the attacker has to invest a large amount of money to succeed. [36] To prevent this attack, [36] [60]:
  - Increase the mining difficulty.
  - Use PoS instead of PoW since it becomes more costly for the attacker.
  - Increase the network nodes.
- **Double-Spending Attack**. It happens when the attacker can perform two different transactions using the same fund by overcoming the consensus mechanism. For this attack to happen, two transactions must be performed before they are verified. There are **4 ways** of performing double-spending. [60]
  - **51% Attack**. Because the attacker controls the majority of nodes, he is capable of manipulating transactions in his favor. In this scenario, the attacker first creates a transaction, TX1, that uses his funds to get another coin or good. In parallel, the attacker will mine a **private malicious chain**. The attacker will start mining blocks and not broadcasting them to the network. In this malicious chain, TX1 (that used attackers' funds) is not added. Because the attacker owns the majority of the computing power, his **private chain** will **grow faster** than the honest one. When the attacker decides to broadcast his private chain to the network, if the "Longest Chain Rule" is applied, all honest miners will drop the **honest chain** and will start mining to the malicious one. Because the attacker's transaction was not added to the malicious chain, the network treats his transaction as if it has never happened, allowing the attacker to spend his funds again. [36]

- **Race Attack**. The attacker makes two mismatched transactions, this is, the attacker creates two transactions with the same funds simultaneously. [131] The target in this attack is any node that accepts transactions with no previous confirmations - transactions that, even though visible, have not yet been included in a block. [23] Therefore, the attacker creates a fraudulent transaction and sends it to the node, and a legitimate transaction and sends it to the network. The node accepts the unconfirmed transaction and sends the goods to the attacker, while the legitimate transaction succeeds, resulting in a double spending attack. [60] [23] To prevent this attack:
  - Nodes should wait for a minimum number of confirmations before considering a transaction valid.[23]
- **Finnney Attack**. The attacker starts by creating a transaction, TX1, that transfers funds to one of his accounts. Afterward, he adds this transaction to a **block**, mined and kept private to the attacker. In parallel, the attacker creates another transaction, TX2, which has the same funds as TX1, but this time, it sends it to a Merchant through the network. This second transaction was not confirmed by any other node and was accepted by the merchant, who thinks it's valid. As a consequence, the Merchant sends the goods to the attacker. Once the attacker receives the goods, he broadcasts the pre-mined block containing the transaction TX1. Because TX1 is now confirmed, TX2 gets invalid because the attacker's funds were already used in the first transaction. This resulted in a double-spending attack because the attacker could get the goods from the merchant and keep his funds. [60] This attack targets the merchant who does not wait for transaction confirmation. To prevent this attack:
  - Nodes should wait for a minimum number of confirmations before considering a transaction valid [23].
- **Vector 76 Attack**. The attacker combines the Race and the Finney attacks. This **targets** nodes requiring only a single confirmation. The attacker established a connection to the victim, E-Wallet, and to a mining pool. The attacker creates one transaction containing a large Bitcoin amount, sending it to a mining pool. When the attacker finds a block, the fraudulent transaction is added, and both the block and the transaction with a small amount of bitcoin are sent to the network simultaneously. The attacker withdraws many bitcoins once the E-Wallet service sees the fraudulent transaction. Meanwhile, the rest of the network is more likely to validate the small transaction, invalidating the large fraudulent one, leading the E-Wallet to lose money. To prevent this attack:

    - Nodes should wait for more than one confirmation before considering a transaction valid. [23]

- **Selfish Mining Attack**. In a regular mining process, a miner immediately validates a block and broadcasts it to the network. However, when a malicious miner uses selfish mining, he mines blocks and keeps them to himself, creating a **private chain**. Once the malicious miner has enough blocks in his private chain, he broadcasts them to the network. If a **fork** is created and if the chain uses the "**Longest Chain Rule**", then the attacker chain will be added to the blockchain, and the attacker will win the blocks and, consequently, its reward. On the other side, honest miners will lose the deserved reward and computational power [61]. This can also be called **block race**, where selfish and honest miners try to win a block. Every blockchain that enables miners to create new blocks is opened to selfish mining attacks. [73] [51] Furthermore, a fork between two selfish miners with big private chains can delay the network, opening the possibility of further attacks. [51] To prevent this attack:
    - Implementing block random selection methods so that miners have a hard time predicting when they will have the opportunity to mine the next block [61] [12].
    - In a fork situation, choose the block with the latest timestamp [61].
- **Sybil Attack**. Attackers create multiple false network identities to flood the network, gaining the majority in the consensus mechanism and, consequently, being able to manipulate transactions. [59] Early networks faced this attack, but stronger and more recent blockchains are more resistant. Furthermore, this attack is not possible in permissioned blockchains, where, because identities are necessary to join the networks, attackers cannot forge identities.[68] To prevent this attack[63] [86]:
    - Raise the cost of creating an identity.
    - Control nodes and detect which are the forwarding blocks.
- **Wallet Attack**. Frequently, blockchain users overvalue security, leading them to be the weakest vulnerability point. Attackers aim to get wallet credentials, and once they get the users' private key, they can interact with the victim's money by executing transactions on the users' behalf. To get wallet credentials, various techniques are used, in between them **dictionary attacks**, **vulnerable signatures**, and **phishing attacks**. [131]
  In **Dictionary Attacks**, a hacker tries to guess passwords using brute force by running them through a list of popular words. The prevention of these attacks can be achieved by using human-only solvable puzzles. [131]

In **Phishing Attacks**, attackers try to get users' credentials through fake e-mails or websites. The goal can be to make victims send money, disclose sensitive information, or download malware through techniques such as manipulation or forgery. There are Social Engineering Schemes and Technical Schemes. Prevention passes to install an anti-malware product, install a browser extension, and verify the source links and sender before interacting. Furthermore, private keys should be stored in hardware wallets or any other offline place to avoid stealing keys.[4]
Vulnerable Signatures, refers to the fact that vulnerable hashes might be used to generate private keys. These attacks can be prevented using an Interactive Incontestable Signature - a signature method proven to be a reliable security mechanism. [131]

- **Target DDoS Attack**. It consists of flooding the network with numerous requests beyond what it handles, making it unresponsive or temporarily inefficient. If the attack succeeds, it might enormously impact the blockchain since users cannot perform transactions or mining. Furthermore, it can also lead to centralized mining, where a few nodes aim to break the blockchain principles. This attack **aims** to saturate the communication link bandwidth, system RAM, CPU resources, or exhaust memory capacity. To prevent this attack[17]:
  - Implement techniques like traffic behavior-based methods, anomaly-based traffic analysis, machine learning, and deep learning.
  - The target victims must monitor their edge infrastructure or content delivery network.
  - Improve blockchain code, filter attack packets, make transactions more expensive, reject if the transaction has confirmed parent transactions, and deploy network relay nodes at the Autonomous System (AS) level.
- **Length Extension Attack**. Attack based on the predictability of hash functions. In this attack, the attacker can add data to the original message by knowing its hash value and length. The attacker doesn't need to know the message content. Performing the Length Extension Attack allows bypassing security checks and creating fraudulent transactions.[23]
  To prevent this attack[119]:
  - Use SHA-25 or Keccak-256, instead of MD5 or SHA-1, which is not vulnerable to these attacks
- **Hash Collision Attack**. A hash collision happens when two different inputs have the same hash value. This might happen because hash functions have an undefined input length and

a predefined output length, making it inevitable for different inputs to have the same hash value. Any program that compares hashes is suitable for this attack. [95] To prevent this attack[23]:

  - Use a strong hash algorithm such as SHA-256.

- **Smart Contracts Exploitation**. Attackers can manipulate smart contracts and expose sensitive information or allow unauthorized access. The solution is to apply good security development practices and implement monitoring tools to track contract activities and unusual behavior. Some of these attacks:

  - **Reentrancy**. The attacker repeatedly calls a function before the previous call has been completed. A contract execution is put on hold when a call to an external contract is performed. This second contract starts having control over the next sequence of events. As such, if contracts' variables are only updated after the external contract call, the validation to execute the method might continue to be true, and consequently, the external contract can continuously keep on calling this method, leading to a loop. If the code being executed is, for example, the withdrawal of funds, this can be continuously done until no more money is left. A preventive measure is finishing all internal contact work (updating all contracts' variables) before calling the second contract.
  - **Timestamp Dependence Vulnerability**. Each block in the blockchain has a timestamp. Miners are capable of manipulating the value of the timestamp. Therefore, "block.timestamp" usage should be avoided; instead, block numbers should be used as they are more predictable and less prone to manipulation.
  - **Integer Overflows and Underflows**. This vulnerability results in surpassing the fixed range of values. The integer type uint8 has an interval of values between 0 and 255. If a variable instantiates with this type surpasses 255, the value rests at 0. And if the variable goes lower than 0, the value rests at 255. This can lead to unexpected modifications in the state variables.
  - **Short Address Attacks**. This attack happens when the address field is not correctly verified. If a user inserts an address shorter than 20 bytes, the Solidity contract will automatically pad with the right zeros to ensure the address is appropriate. A prevention for this attack is to use **bytes20** data tupe for address, which ensures the fixed-length.

These are just a few examples of attacks that can happen on the blockchain. Besides the ones presented, many others can be performed.

1.1.4 Biggest Problems with Blockchain

Many problems are hindering the adoption of blockchain. They are scalability, energy consumption, security, complexity, and interoperability.

- **Scalability**. As the number of blockchain transactions increases, the difficulty of validating them also increases, leading to transaction throughput limitations and transaction fees. This can be considered when developing solutions requiring fast transaction processing speeds. Solutions to overcome this issue pass by including off-chain channels. [68]

- **Energy Consumption**. Some consensus protocols, like PoW, require a lot of computing power, which increases carbon emissions affecting the environment. As a solution, blockchains like Ethereum have adopted other consensus protocols like PoS, which consume less energy. [98]

- **Security**. As previously seen in section 1.1.3, blockchain still presents many attack vectors. Security efforts need to be performed to ensure stronger security in blockchain. These include implementing good security practices and monitoring mechanisms to track unusual behavior.

- **Complexity**. Blockchain is a complex technology that implies a lot of technical knowledge. This might hinder its adoption, discourage potential users and developers, and lead to errors and inefficiencies in implementation. A solution to this passes in developing user-friendly solutions and promoting educational resources. [98]

- **Interoperability**. Interoperability between networks is the ability of different blockchain networks to communicate. Because each network has its own protocol, it's still a challenge to make them collaborate together. This lack of interoperability leads institutions to navigate through multiple platforms, tokens, and cryptocurrencies, leading to a bigger effort. Furthermore, this might prevent innovation and development of solutions that might benefit the end user. [103]

- **Privacy**. It's possible to discover transaction patterns and establish a relationship between them. Imagine a user using the same address for more than one transaction. In this case, tracing them back and discovering the users' historical and financial activities is possible, leading to identifying the entity. [134] Furthermore, through attacks like man-in-the-middle and DoS/DDoS attacks, transactions with confidential content can also be compromised [103].

- **Regulatory Compliance**. Applying blockchain technology in non-Bitcoin currency's legal and financial sectors creates regulatory challenges. Blockchain lives independently of the law, and government officials won't change their behavior based on the presence of this new technology. [131]

To achieve the full potential of this groundbreaking technology, the listed problems will have to be overcome.

1.2 dApps

Decentralized Applications, dApps, are applications built on a P2P network, not depending on a central authority to manipulate the data or actions performed on the application. The communication channel is the blockchain, where each interaction in the dApp generates an entry in the blockchain, in which data gets stored using cryptography. [45] [10]

Normally dApps are developed under the Ethereum blockchain, which provides **smart contracts** - self-executing programs - enabling peer communication and execution when certain conditions are met [3]. These smart contracts are responsible for storing the application logic and can be seen as classes in centralized programming. [45] Since dApps are developed over a blockchain and use smart contracts to execute functionality, they can be considered persistent.

By using blockchain, dApps **inherit** their characteristics, and users maintain absolute control of their data at any time. Even though dApps resist censorship and provide privacy, they are in their early stages, facing **problems** and challenges such as scalability. Furthermore, because smart contracts cannot be changed once deployed, it's important to ensure their well-functioning before deploying them. [45] [126] [1]

1.2.1 dApps security

When developing a dApp, security must be taken into account. Not taking it seriously can lead to severe consequences, such as loss of sensitive information, financial assets, and user trust. Because dApps are developed over the blockchain, they inherit not only all their security features but also all their vulnerabilities and weaknesses.

When developing a dApp, some points should be taken into consideration:

- **Data Integrity and Confidentiality**. Cryptographic techniques with strong encryption algorithms should be used to protect sensitive data from tempering and unauthorized access. Asymmetric encryption and a secure hashing mechanism can be used. More details on them in 1.3 and 1.4. Only the necessary information should be added to the smart contract, and its management should be carefully maneuvered.

- **Principle of Least Privilege**. Users should only have access to what they need to fulfill their tasks. This reduces the attack surface and minimizes the impact of security breaches.
- **Smart Contracts Robustness**. Smart contracts, once deployed, become open-sourced and, if left vulnerable, are susceptible to hacking.
- **Blockchain selection**. The chosen blockchain should fulfill the dApp requirements and present characteristics, such as a consensus mechanism that is robust and resistant to attacks.
- **Scalability Challenges**. As people adhere to a dApp, scalability becomes a concern. Increased usage may lead to slower transaction processing. These delays can latter be exploited by attackers.

1.3 Encryption

Encryption is a **reversible** process used to ensure the **security** of data by **encoding** it. This mechanism ensures confidentiality and controlled data access. There are several types of encryption: symmetric, asymmetric, hybrid, stream, block, homomorphic, quantum... In this section only symmetric and asymmetric are going to be analyzed.[133]

Symmetric Encryption: uses the same key to encrypt and decrypt, and therefore, this key should be kept secret in order to enhance security. Frequently used in circumstances where speed and efficiency are required. It is used to encrypt large amounts of data. Furthermore, this type of encryption uses Initialization Vectors (IVs), a pseudo-random value used to ensure unique encryption outputs, even when the same key and plaintext are used. This IV is typically used in block ciphers, such as Cipher Block Chaining (CBC), Counter (CTR), and others. Examples of symmetric encryption are AES (Advanced Encryption Standard), and DES (Data Encryption Standard).[133] [26]

Asymmetric Encryption: uses a key pair formed by a public key and a private key. While the first key can be shared and used to encrypt data, the second key can not be shared and is used to decrepit data. Examples of asymmetric encryption are Rivest-Shamir-Adleman (RSA), and Elliptic Curve Cryptography (ECC). Often used to secure key exchange, digital signatures, and encryption in Public Key Infrastructures (PKI).[133]

1.4 Hashing

Unlike encryption, hashing is a one-way process, this is, once a plaintext has been hashed, it's not possible to get its value back. Hashing creates a unique, fixed-length output from an input of any size. Changing something on the input will generate a totally different output. It is commonly used for integrity verification, digital signatures, password storage, and other applications where irreversible transformation is needed.[53] There can be:

- **Cryptographic Hash Functions:** designed for secure applications like digital signatures, password storages, and data integrity checks. Examples of it are SHA-256, SHA-1, SHA-3, and MD5. Where SHA-2 and SHA-3 are considered extremely secure. [133]
- **Non-Cryptographic Hash Functions:** used when security is not a priority, focused on speed and simplicity. Used in data structures (hash tables databases, cache systems...). Examples of it are MurmurHash, CityHash, FNV. [53]
- **Checksum Algorithms:** simple algorithm, used for error detection and correction in data transmission. Not collision-resistant, but rather used to detect data changes. Examples of it are CRS, Adler-32. [78]

1.5 Hierarchical Deterministic Wallets (HD)

Frequently, when we talk about wallets, we also talk about key generation because owning a private key is the same as owning coins, and for so, a wallet and a key have similar meanings.

The MetaMask application is a wallet that uses the concept of Hierarchical Deterministic Wallets (HD). As such, MetaMask provides a **seed phrase**, also called mnemonic, that enables users to access their accounts. Because of this, it is important to keep the mnemonic secret.

This seed is represented as a mnemonic phrase following the **BIP-39 standard**. For the mnemonic generation, a source of entropy (randomness) is typically generated as a 128-bit (originates a 12-word mnemonic) to 256-bit (generates a 24-word mnemonic) random number. This is followed by a checksum calculation concatenated with the entropy to form a new binary sequence. This binary sequence is then divided into 11-bit segments. Each of these segments is then mapped to a word from a wordlist of 2048 words, defined in the BIP-39 standard. The result is a human-readable phrase.

The **security of the mnemonic** relies on the quality of the entropy used. If the entropy is truly random, the resulting mnemonic will be secure and difficult to guess. Additionally, the wordlist contains 2048 unique words, creating a vast number of possible word combinations.

This significantly complicates attempts to guess the mnemonic through brute force.

To be able to generate the **same set of keys for the same seed, this is, mnemonic**, the Hierarchical Deterministic Wallet is based on the **BIP-32 standard** and **BIP-44 standard**. This key generation is performed using **Elliptic Curve Cryptography (ECC)**. Furthermore, the generated private key is kept in the user's local storage, regenerating every time the user changes the device or cleans the local storage.

With this being said, it is possible to summarize that a seed phrase provides a simpler way to memorize a very long number (seed) later used to generate accounts deterministically. [66]

In computer science, deterministic describes a process that will always generate the same result. In this case, the same mnemonic will always generate the same set of keys/accounts. [66] It's important to refer to the difference between Deterministic Wallets and Hierarchical Deterministic Wallets (HD):

- **Deterministic Wallets:** Refers to a wallet capable of generating multiple private keys from the same seed. Frequently uses SHA-256 hash function.
- **Hierarchical Deterministic Wallets:** Unlike deterministic wallets that generate one private key, in hierarchical deterministic wallets, a pair of master keys are directly generated from the seed that was previously generated from a mnemonic phrase. From here, it's possible to derive a tree of cryptographic keys.[93]

As previously said, this process uses different protocols, such as BIP39 and BIP44, which have different purposes.

- **BIP39 - Bitcoin Improvement Proposal 39:** widely used open-source protocol capable of generating a seed phrase and converting this phrase into a seed, which is then used to generate Hierarchical Deterministic Wallet (HD) keys using schemes like BIP-32 or BIP-44. The generated seed phrase allows users access to their wallets, and they come from a predefined dictionary of 2048 words (from multiple languages) that are then randomly chosen (similar words are avoided), guaranteeing that each seed phrase is secure and hard to guess. [55]
- **BIP44 - Bitcoin Improvement Proposal 44:** defines hierarchical deterministic paths from different currencies, accounts, and addresses. It doesn't provide a way to generate mnemonics or seeds (such as BIP-39), but it describes the structure of the derivation

path and its hierarchy. The image in appendix B shows how it's possible to use BIP-32 to deterministically generate a sequence of Bitcoin wallets using a seed phrase.[55]

In the image presented in appendix E.4, it's possible to see a code snippet explaining how the account generation is performed. In this snippet, there is a "path" argument. This argument is how multiple accounts are generated from the same mnemonic. Each "/" is considered to be "path levels" where [93]:

- **'m':** root of the hierarchy.
- **'44'':** indicates that follows the BIP-44 standard.
- **'60'':** in BIP-44 each currency is assigned with a unique number. The number 60 is the one for Ethereum.
- **'0'':** account index, in this case, the first account.
- **'0'':** change addresses, where 0 is for external addresses and 1 for internal/change addresses. Doing m/44'/60'/1'/0/0 => creates a second account.
- **'0'':** address index.

As for so, the "path" argument describes how to reach a specific node through the hierarchical structure, that is, to reach a specific cryptographic keypair. [93]

1.6 Authentication

Authentication is about verifying the user's identity. While in web2, this is performed using a username and a password, in web3, this is done using a mnemonic, as explained in the previous subtitle (1.5). [56]

1.7 Authorization

Authorization is about giving permissions to a user who has already been authenticated with an identity.[56] In web3, this can be done by passing the user's Ethereum address as part of the request payload. When the smart contract receives the request, validations are performed to ensure that the user can execute the functionality. This can be seen in the code snippet in appendix E.7, where to execute the "editFile()" method, the transaction executor can only be the contract "AccessControl.sol".

1.8 Session Timeout

Session timeout is a mechanism that disables a session after a period of inactivity, as such preventing a session from being active indefinitely. Once the session has expired, the user has to be re-authenticated. This ensures that only authorized users maintain access to sensitive resources. [33]

While in web2, session timeouts are managed using session tokens or cookies, in web3, session timeouts are handled using cryptographic keys or wallet addresses.

This session timeout helps to mitigate risks:

- **Reduce exposure to unauthorized access:** If a user is hijacked - attacker takes unauthorized control over a website - or compromised, the impact is limited to the expiration time. The attacker would need to re-authenticate once the session expires. [33]
- **Mitigates shoulder surfing:** shoulder surfing refers to the act of observing someone's activity to get sensitive information such as passwords or account numbers. Therefore, someone else might view or manipulate sensitive data if a user leaves a device unattended. [114] Session timeout allows automatic logout if a period of inactivity is reached.

1.9 Dissertation Structure

This document is organized into eight sections, concluding with References and some Appendices. To enhance the clarity and structure of the document, all visual materials, including images, schematics, flowcharts, and tables, are shown in the appendix. Each section begins with a concise introduction outlining the topics to be discussed, followed by the main content and a summary of the key conclusions.

This first introductory section provides essential background information. After this, the document includes a Literature Review, which discusses the identified problem, proposed solutions, relevant scientific research, and existing market solutions. A comparative analysis of potential technologies precedes a detailed exposition of the selected technologies, specifying the solutions' objectives, requirements, use cases, and architecture. The following sections describe the implementation process and the results achieved. And lastly, the final chapter offers a comprehensive summary of the conclusions drawn and the outcomes obtained.

Chapter 2

Literature Review

This section starts by contextualizing the reader on the problem and solution addressed in this dissertation, followed by a market analysis and a review of scientific papers found in bibliographic databases such as IEEE, Web of Science, Springer Link, and Scopus.

Furthermore, this section compares the proposed solution with existing solutions found in scientific literature and the market, explicitly identifying the innovation proposed by this dissertation.

Finally, it outlines the research questions, which are previously answered in the conclusion, chapter 8. It also proposes a validation strategy and discusses potential threats to the validity of the findings.

2.1 Problem/Context

In recent years more and more documents, such as birth certificates, and university diplomas have been stored in digital form due to technological evolution. This was seen as a way to achieve a paperless environment and as a way of helping in spreading information. This transition enables easy access to documents, but it provides an open door to easy **document forgery** or **counterfeiting**, allowing individuals to get credit for undeserved accomplishments. [19] [96] Furthermore, it's possible to perform identity document forgery and steal someone else identity. [8]

Sectors such as financial sectors are now at their peak concerning fraud and document falsification. [110] A common case might happen when the client receives a document containing all the savings account information. Because the bank institution only stores the data and not the signed document, the client can change information, such as the interest rate, and sue the bank

for trying to deceive him. As in this case, many other sectors, such as universities and hospitals, face this problem. As such, there is a need to have an efficient way of solving this issue.

Reference [96] describes four techniques already used to ensure/prevent document alteration: going through Barcode, Hashing, Document Signatures, and Watermarks. Alongside these techniques, industries provide solutions to detect or avoid falsified documents, such as **human analysis**, **AI** (Inscribe [42], Nethone [72], resistant.AI [92]), and **document-sealing solutions** (Entrust [27]). Nevertheless, they are either not efficient, in the case of human analysis, or centralized.

Furthermore, due to forgery issues, companies/institutions have been adding **complex processes** to ensure document verification [96]. An example is reflected in the recruitment process. Because it's possible to counterfeit diplomas, companies introduced competency tests in the interview process since the diploma is no longer enough to ensure an individual's knowledge. [77]

As it will be analyzed in the sections to come, **there is no solution or scientific paper that develops a document management, tracker, share, and verifier system oriented to decentralization over security**.

With this being said, the dissertation aims to address two main problems:

- Avoid document **counterfeiting** and **forgery** in different sectors.
- Avoid **long processes** created as a solution for document counterfeiting and forgery.

2.2 Solution

In order to address the problems previously explained, a blockchain-based system, **oriented to decentralization** is proposed. The solution is developed to be a file manager, also enabling the tracking, sharing, and verification of files. As so, **features** such as file uploading, deleting, and sharing, alongside an audit log will be provided by the decentralized application. With these features, it's possible to ensure a single place where documents can be edited and issued, always guaranteeing their **security** and **trustworthiness**.

For the solution proposed by the current dissertation, the **end-users** are to be individuals in a **public environment**. Nevertheless, in the future, the solution can evolve and be used by companies/institutions, enabling secure document circulation inside them.

This solution aims to be a single source of truth, where citizens can verify if a document has been changed (forged) and track every action taken against it. The **main focus** is to ensure decentralization in document management, issuance, tracking, sharing, and verification.

Furthermore, this solution will be used as a starting point to explore how can security and decentralization be **balanced** to promote robustness and user trust. It also allows for a comprehensive **examination** of the challenges when developing with blockchain.

2.3 Related Work

Much work has been done concerning using blockchain to avoid document forgery and to study the use of blockchain as a secure way to store and share files. This section aims to showcase the scientific papers related to the problem previously explained alongside the market study.

2.3.1 Scientific Papers

Reference [7] presents a survey on decentralized applications based on **Ethereum**, the use of smart contracts, **IPFS** (InterPlanetary File System), and other tools and techniques used to develop dApps on the Ethereum network. The article stated how hard it is to maintain and develop a dApp on a large scale and states **traditional security is insufficient** for blockchain-based apps.

Reference [34], addresses the **need** for a decentralized, distributed **data storage and access solution** that is simple and doesn't need financial incentives. To this end, VAULT, a permissioned blockchain-based protocol, is developed and analyzed. It has the characteristics of not needing financial incentives and enabling **features** like adding, updating, and sharing documents between users. To ensure efficiency **IPFS** is used and, to ensure security, a **permissioned blockchain** is used alongside a quorum-based approach, which guarantees fairness through randomness in quorum selection. Furthermore, the files' content is encrypted before being stored in IPFS. VAULT's end users are applications that require collaboration from multiple permissioned parties, such as project management or collaborations among researchers or developers. This article can **conclude** that VAULT is trustless and transparent, solving the problem of a single point of failure presented in the centralized storage solutions. Additionally, they can demonstrate the fairness and linear scalability of the employed quorum selection, enhancing the capacity to handle a greater number of transactions per block.

Reference [120] addresses the **need** for a **file-sharing** system independent of a third party, this is, a decentralized file-sharing system. A blockchain-based file-sharing framework that leverages the advantages of **IPFS** and **PKI** systems is described as a way to solve the mentioned need. **Features** like uploading and sharing files are provided by the solution. **Ethereum**

Blockchain is used, enabling authentication, authorization, and integrity. IPFS is used to provide availability to the system and help lower the transaction cost - heavier content is stored in the IPFS, while lighter content is stored directly in the blockchain. Security is provided by access control privileges (enabled using smart contracts), usage of a modified version of IPFS, synchronous encryption, and usage of Advanced Encryption System (AES) before sending a file to the IPFS. The solution is developed for a **web environment** and uses **OpenPGP** as a PKI. Metamask is used for a smooth login and integration of cryptographical authentication. Furthermore, **Oyente** tool tests vulnerabilities in the smart contracts [76].

Reference [49] addresses the **need** of providing a deep analysis on developing a secure **file-sharing** system based on blockchain and **IPFS**, in order to reduce overheads and security threads. Using blockchain and IPFS without any other security measure becomes unreliable since their content is accessible to anyone in the network. To get around this, the paper proposes to encrypt data using the **AES** (Advanced Encryption Standard) before sending it to IPFS. Furthermore, another security layer is added by using a **permissioned blockchain** since only allowed nodes can interact with the network. Besides the security layer, the article discusses that IPFS also makes the used blockchain more scalable and lightweight and helps decrease the cost of storing data in the blockchain. As a form of **analysis**, it also presents the achieved results related to the computing time when uploading and downloading a file.

Reference [5] provides an in-depth analysis of the existing **file-sharing** solutions and how the existing problems can be accessed using blockchain and **IPFS**, always maintaining privacy and security. The paper provides an **analysis** of centralized and decentralized solutions (Sia, Storj, and FileCoin), and an analysis of the combination of IPFS and blockchain. The paper proposed a system developed using the **Ethereum blockchain** in a **private network**. They proved that, by combining IPFS and blockchain, it's possible to have a secure, fast, and highly available platform for storing and sharing data. The proposed system improved data security and provided faster file retrieval and a higher degree of data availability. While blockchain provides transparency and security, IPFS provides security and efficiency of data storage and sharing. **Limitations** related with **scalability**, **security**, and **compatibility** were encountered in the proposed system.

Reference [64] addresses the **need** for a blockchain-based, secure **file-sharing** system developed for a **consortium of organizations**. A workflow for identity management and file-sharing processes is developed using **Hyperledger Fabric** alongside **IPFS** and Identity and Interfacing Server (**IIS**). It provides **features** of identity registration, user identity authentication, file

sharing, and file retrieval. The proposed system provides security, confidentiality, integrity, and availability of shared files.

Reference [46] addresses the **need** for a system that helps **educational institutions** in record keeping/document management, decreasing problems of losing files and avoiding waste of time and resources. A secure and public **Ethereum** blockchain-based file-tracking system that ensures data governance and privacy of college students is proposed. It uses the **Chameleon Hash Function** in conjunction with the Advanced Encryption Standard (**AES**) algorithm. Chameleon Hash Functions provide an additional layer of security by ensuring documents are safely delivered and protected by authorized access. **AES** is an algorithm to protect user data and prevent unauthorized access. Furthermore, the developed system provides file **upload, tracking, and management** features alongside **access control, audit trails, and secure sharing of files**. The proposed scheme provides a secure, scalable, and transparent solution through sophisticated security features and advanced cryptographic techniques.

Reference [79] addresses the **need** for a decentralized **private storage** solution with an efficient file management system to help **file sharing** between organizations. A storage system developed using **Hyperledger**, a consortium blockchain, is suggested, and tests are performed to ensure the systems' feasibility. As for **features**, the system enables file storage, identity authentication, access control, file retrieval, inserting nodes, selecting files and nodes, deleting files, and adding authorized organizations.

Reference [74] addresses the **need** for a decentralized **version control** where multiple parties interact. A document version control and **sharing Ethereum** application is developed to enable multi-user collaboration in a secure and trustworthy way. **IPFS** is used. The proposed solution enables control of data sharing and version tracking. Furthermore, a vulnerability and security analysis is performed to check for common bugs and vulnerabilities.

Reference [71] addresses the **need** to solve the current centralized process of **tracking medical records**. This centralization introduces uncertainty in stored data integrity, identity theft, lack of accessibility (medical institutions have different data-sharing policies and regulations), and opportunities for malware, viruses, and hacking attempts. In order to solve this need, **Rechain**, a blockchain-based digital medical health **record management** system is proposed. This solution provides security, availability, and integrity being easily accessible from anywhere and stored in one single location. It uses **Ethereum** network, **web3.storage** which is related to **IPFS**, and Metamask. **Roles** of admin, doctor, and patient are used to provide different levels of ac-

cess, and therefore provide security to its stored content. This solution solves the bureaucratic processes to get patient data, takes out the need for a third party, and stores all the content in just one place. Nevertheless, this solution faces **challenges** of latency, costs (every transaction has a fee), and editing records.

Reference [97] addresses the **need** for a secure and highly available platform that holds the **Personal Health Record** (PHR) for patients and healthcare providers. A decentralized, blockchain-based solution is proposed, providing **features** of login, access to patient's health records, view of access list, Gmail access, view patient records, and patient record creation. Using **Ethereum** and **IPFS**, it **stores** health records in a secure way. In order to answer the research question: "Is Ethereum blockchain and IPFS suitable for storing Personal Health Records?" they performed tests, achieving success in all of them. The system was proven to provide interoperability between care systems and immutability in historical medical data. The only limitation of the system is the **transaction cost**.

Reference [98] addresses the **problem** of counterfeiting in the **medicine supply chain** which can impose damage to pharmaceutical companies and a risk to public health. A blockchain multilevel security and authentication solution is proposed. It enhances transparency, integrity, and traceability. It's implemented using **Hyperledger Fabric**, and it incorporates a blockchain-based **QR code** watermarking layer for authentication and verification. **Access privilege restrictions** and multiple validations were implemented, and the system provides three levels of verification of medicines: QR code authentication, verification, and traceability. Furthermore, project performance metrics were measured using **Caliper tool**.

Reference [96] addresses the **need** for a platform that provides the means to **store** documents securely, avoiding problems like **forgery** and **counterfeiting**. A blockchain-based solution incorporated with the use of **IPFS** was presented. It consists of a **web-based application**, programmed in Java and upon a **self-developed blockchain**. In order to guarantee the authenticity, integrity, and nonrepudiation of a document, the solution provides features such as account creation, document upload, document consulting, and document verification.

Reference [91] addresses the **need** of ensuring that forensic reports are **kept private and unchanged** by malicious actors. Because public blockchains are not suitable for preserving the privacy of an organization and since they imply the usage of cryptocurrency to execute every transaction, the authors of the article selected a private permissioned blockchain, **Hyperledger**, alongside the use of **IPFS**. The solution encounters **three different types** of users with different

responsibilities, the end-user, the system admin, and the secondary admin. Accordingly to the user role, it's **possible to** access the register of an analytical report, build/update/remove/fetch, and trace the updates of any forensic report. Furthermore, with the proposed solution it's possible to accommodate both textual and media files. To ensure security the article opts for a permissioned blockchain, uses different roles with different possible actions, and different access levels to a forensic report. The authors of the article claim that the system provides traceability and prevents data tempering from malicious actors. It was analyzed that the encountered solution overcomes traditional public decentralized storage systems, once that **transaction processing time** achieves an average of 11.99 seconds/transaction. Therefore it has a faster transaction execution when compared to other blockchain systems.

Reference [52] addresses the **need** of a blockchain architecture that provides secure transmission of examination papers from the superintendents to the students, eliminating the need for a third party. This process is currently performed physically which incurs many security issues. The paper proposes a blockchain-based system that will eliminate the need for physical circulation. This solution uses **Ethereum** blockchain and **IPFS** to achieve decentralization and secure communication between the teachers and superintendents. Other tools such as **Metamask**, and **Ganache** are also used.

Reference [77] addresses the **need** to ensure legality in documents related to **education**, especially at universities in Indonesia, avoiding problems like diploma counterfeiting/forgery. As a result of these types of fraud, bureaucratic processes in recruiting new employees are introduced, implying a series of tests to ensure the individuals' knowledge. An **Ethereum-based** solution was suggested to help with **document management** in the field of education. The proposed **features** are authentication, publishing, and document verification. The solution uses architectural models, algorithms, tests, and implementation details that are universal and can be used to issue and verify different types of certificates. The paper examines this solution and sees that it reduces time spent validating authenticity.

Reference [94] analyzes how blockchain can be used in **education systems**, avoiding document falsification, such as diplomas. Even though many blockchain-based solutions have been proposed, none of them were able to be widely used, and there is no recent research that shows their usage once the article was published. This reference analyzes blockchain in the education sector, showcasing limitations and advantages alongside possible implementation models.

Reference [31] addresses the **need** for a decentralized system that prevents document forgery

in the **education sector**, which also leads to lengthy document verification processes. An **Ethereum** document verification decentralized **web application** was proposed. With the system, it was possible to achieve scalability and latency without opening the hand of simplicity, minimalism, and cost-effectiveness.

Reference [47] addresses the **problem** of fraud in **real estate contracts** and the scalability issue that comes with blockchain usage. An **Ethereum** online real estate application was developed using a **zero-knowledge proof algorithm**. It enables online contract management and discrimination of contract forgery. It ensures full reliability without the need for intermediaries.

Reference [130] addresses the **problem** of lack of security and anonymity in **e-voting**. Because encryption alone can not guarantee anonymity, trying to develop a more secure system has become an important matter. A blockchain solution was developed to prevent the forgery of votes, being considered public, distributed, and decentralized. An **Elliptic Curve Cryptography (ECC)** was used to provide authentication and non-repudiation, and a withdrawal model was designed to enable voters to change their minds until a deadline is achieved. The solution enables auditing and verifying votes without voting. It was proven to be a practical and secure e-voting system to avoid votes' forgery.

Reference [21] addresses the **problem** of document forgery which, consequently, provides strict verification and authentication procedures to guarantee documents are original. As a solution, an **Ethereum** document verification system is proposed and analyzed, developed to be used by multiple organizations, and used as a **generic platform**. It's developed to be **decentralized**, **permissionless**, and open to everybody. Each organization has its system, run by an **admin** which is able to select **verifiers** responsible for checking documents uploaded by users. To become an admin a money transaction has to be performed. The system also uses **IPFS** to store documents. They provide document version control using IPFS and security through an admin role and intend, in the future, to add features as document categories in order to provide documents with their specific sector fields. As for **functionalities**, it enables document upload and storage, add verifiers, and document verification.

Reference [13] addresses the **problem** of fake **passports**. An **Ethereum** decentralized application for managing passport-related information is designed and implemented, enabling the verification of a passport's authenticity. It is fast and secure and uses **IPFS** to solve the blockchain storage limitation. It enables **storage** of passport details, passport data retrieval, report dashboard, and alert notification when a passport is forged.

Reference [69] addresses the **problem** of forgery of black box **image data**, which helps to determine responsibility in traffic accidents. By developing a blockchain-based solution, using encryption, **IPFS**, and developing a **self-validating consensus algorithm**, the article aims to solve the data privacy and storage problem of blockchain as well as the consensus algorithm latency problem and the forgery of black box image data. The proposed solution was analyzed in terms of performance and concluded the downloaded and uploaded don't affect the performance, and there is a linear increase in the delay time according to the capacity.

Reference [90] addresses the **need** to avoid counterfeiting of **digital certificates**. For so, a **Hyperledger** low-cost and private blockchain is suggested. It has an admin that maintains control over systems' data, has access to historian records, updates and creates them, and customizes new participants. Furthermore, it also has an end user, which has various restrictions on document access, modification, and creation. The solution provides a feasible, secure, and stable way of managing certificates in a private network.

2.3.2 Market Study

Besides scientific papers, this section describes some of the already existing market solutions.

Decentralized Identity Providers solve the centralization problem in authentication and verification, enabling greater control and privacy. It's formed by DIDs, decentralized identifiers, and VCs, verifiable credentials, which provide aspects to the user identity. [15] VCs hold information such as birth date or the degree of study of an entity. An example of a decentralized identifier is PolygonID. [9]

File Management Systems can be centralized and decentralized. Cloud storage systems, such as "Microsoft OneDrive," "Samsung MyFiles, "Files by Google," and "Dropbox" are centralized and enable the management of documents (store, share, delete, and more). Decentralized storage systems, such as "Storj" [111], "Sia" [101], "FileCoin" [30], "Swarm" [116], and "Blockhouse" [14], also use blockchain to provide a decentralized way of storing documents.

Cryptowerk is a platform that helps with file verification at a massive scale. It leverages the benefits of blockchain technology to mitigate the risks of file tampering. It provides an API that communicates with the blockchain and returns the veracity of a document. It provides endpoints to store documents' hash on the blockchain, get the documents' seals, and verify seals.[107]

CB Blockchain Seal, developed by Connecting Software in collaboration with CryptoWerk, is a solution designed to ensure data integrity in platforms like SharePoint and Salesforce. CryptoWerk's approach involves storing only the hash of the files on the blockchain, ensuring that Connecting Software never stores any original customer data on the chain. CB Blockchain Seal accumulates the users' requests for a more efficient solution before sending them to the blockchain.[22]

DocuTrack is an open-source document-sharing dApp developed on the Internet Computer Protocol (ICP) that allows sending and receiving documents in a fully encrypted way. A user can upload a file and generate a secret key for that file. Afterward, the user selects a recipient to decrypt the files' secret key and access its content. All of this without providing personal information. It also allows the recipient to make a file-sharing request.[25] [65]

Pinata is an IPFS pinning service. It enables uploading, editing, sharing, deleting, and renaming files. It also provides an API that enables more control over how files are used.[83]

2.3.3 Comparison With The Proposed Solution

The proposed solutions **can be divided** into file sharing and storage, as well as file counterfeiting/forgery into broad and specific areas. This sub-section aims to detail each of these and compare them with the proposed solution.

2.3.3.1 Counterfeiting/Forgery in broad areas

- **Reference [96] -** The solution is seen as a document verification system. It uses a permissioned self-developed blockchain, and for so it abdicates decentralization to provide more security by being able to choose who joins the network. Because it is "permissioned", the presented solution is developed to be used in a closed environment, inside of an institution/entity.

- **Reference [90] -** Also addresses the problem to be solved with a permissioned blockchain, giving more priority to security over decentralization.

- **Reference [21] -** The solution is seen as a fully decentralized document verification system. It has roles of admin, verifier, and user and enables uploading, storing, and document verification. The main users are organizations. Specific features such as admin and verifier are present to the final users.

- **Proposed Solution -** The proposed solution is a system used to manage, track, share, and verify files, not only being used for document verification. It aims to be used in an open

environment prioritizing decentralization and ensuring security using other mechanisms besides the ones provided by permissioned networks. The main users are individuals. A limitation of the solution is that every transaction will have a cost but this will be compensated using IPFS and other processes explained in sections to come. This solution is seen as a document verifier and manager. Furthermore, because the blockchain to be used is public, the system to be developed sacrifices extra security provided by private blockchains in order to get full decentralization.

Lastly, reference [96] only presents how a file can be stored in the blockchain securely, not providing insights on best practices, tool comparison for dApp development, and efficiency measures.

2.3.3.2 Counterfeiting/Forgery in specific areas

The problem and mitigations of counterfeiting and forgery of documents in specific knowledge areas is already being addressed by the following articles:

- **Reference [91]** addresses the problem in forensic scenarios.
- **Reference [71], [97], and [98]** addresses the problem of counterfeiting and forgery in the medicine sector.
- **Reference [47]** addresses the problem of forgery and counterfeiting real estate contracts.
- **Reference [130]** addresses the problem of counterfeiting and forgery of votes in e-voting platforms.
- **Reference [77], [46], [52], [94], [31]** address the problem of document forgery and counterfeiting in the education sector, developing different systems with specific features and with a deep analysis on it.
- **Reference [13]** addresses the problem of forgery and counterfeiting in passport falsification.
- **Reference [69]** addresses the problem of forgery and counterfeiting in-vehicle image data.

With this being said, while the presented articles developed features, transactions, and security measures focused on a specific use case/area of study, the proposed solution aims to be broad and enable storage, management, and verification of any existing document.

Furthermore, articles [121], [35], [128], and [7] don't aim to directly solve the counterfeiting of documents but to show that blockchain is capable of simplifying specific processes.

2.3.3.3 File sharing, and storage solutions

Reference [96], [64], [5], [49], [120], [34], [79], [74] are related to file-sharing, data storage, and access solutions. Nevertheless, file-sharing is just a feature of the proposed dissertation solution. These references don't present the verification feature, which is fundamental to avoid counterfeiting/forgery, and they don't focus on the need to be addressed by the dissertation.

Lastly, the article [7] focuses only on the analysis of applications developed on Ethereum blockchains alongside IPFS. Contrarily, the dissertation addresses the specific need of counterfeiting, and forgery, and provides an analysis of best practices, tools, and security measures on dApps.

2.3.3.4 Market Study - Decentralized Identity Providers

Both solutions, such as PolygonID and the one being developed in the dissertation, aim to provide trustable information. Nevertheless, NearFile, goes further beyond, it aims to solve the problem of information disclosure, which is not addressed in Polygon ID. Furthermore, NearFile also works as a file manager with a document verification feature, which is not one of the features of Polygon ID. NearFile aims to be much more than an identity provider, it aims to ensure trustability in all types of files, not only the ones related to the user's identity.

2.3.3.5 Market Study - File Management System

The presented solutions only serve as file providers, not ensuring the validity of documents, nor are they used as a place to store and track authentic documentation.

2.3.3.6 Market Study - CryptoWerk

CryptoWerk primarily focuses on decentralized file verification, but the proposed solution goes further by decentralizing verification, file storage, management, and sharing, along with providing audit logs. Unlike CryptoWerk, which is limited to an API, the proposed solution offers a comprehensive file management platform. However, it's worth noting that the proposed solution can still leverage CryptoWerk for its file verification capabilities.

2.3.3.7 Market Study - CB Blockchain Seal

CB Blockchain Seal only extends the usage of Cryptowerk to SharePoint and SalesForce. As for so, it is also focused on decentralizing file verification, not contemplating other features of the proposed solution.

2.3.3.8 Market Study - DocuTrack

DocuTrack, provides a platform that focuses on the decentralization of file sharing. Consequently, it also doesn't provide all the features the proposed solution aims to possess.

2.3.3.9 Market Study - Pinata

Pinata is an IPFS pinning service, so it focuses on easing the process of users storing files in the IPFS. It doesn't use blockchain for this file management, and it doesn't provide audit logs and file verification.

2.3.4 Innovation

Because the **need** being addressed by the dissertations' proposed solution is already analyzed by articles like [96], [77], and [21], it's possible to conclude that its innovation **is not** about solving an unbreakable problem/necessity in the market, but **it's about** how that problem/necessity is being approached.

Even though there are applications or scientific papers that separately implement some of the features proposed by the solution, they do not develop them all while prioritizing decentralization. Furthermore, they don't provide an analysis of good security and development practices as well as how development, including testing, was performed.

The dissertation aims to provide a solution that can be used in a **public environment**, **not use case-specific** (unlike articles [71] or [77]), and **prioritize decentralization**. The dissertation aims to furthermore analyze how can decentralization and security be hand in hand and understand possible drawbacks and benefits of developing apps in the blockchain. It also aims to provide an analysis on good security and development practices as well as testing.

The **research questions** are:

- Question 1: "Is it possible to develop a secure, 100% decentralized file manager, tracker, share, and verifier system?"
- Question 2: "How can a dApp that aims to store important and personal files be public, secure, and decentralized?"
- Question 3: "What are the systems' limitations, and how do they affect the end user?"
- Question 4: "What are the best security measures when developing a dApp?"
- Question 5: "What are the best tools and practices when developing a dApp?"

The next section proposes ways in which these questions are going to be approached/answered.

2.4 Validation Strategy

This section aims to explain the process to be performed in order to achieve the needed answers to the research questions. Accordingly, with the proposed solution, the performed tasks were:

- Study, and **compare** various **technologies** and **security** measures.
- Development of a **proof of concept**. Where various areas of software engineering will be taken into account: requirements management, implementation of software and architectural patterns, as well as security and testing coverage analysis.
- Development of **unit tests**.
- Usage of **security and testing tools** to analyze the robustness of the developed solution. In order to understand and extract quality measures of the proposed proof of concept.

2.5 Threats to validity

Threats to validity refer to anything that might compromise the conclusions taken from the dissertation development, being important to identify them from the beginning. Three threats to validity were identified:

- It's **not feasible to read all** the existing articles in the scientific community. For so, some important articles might not be considered.
- **Technology evolution**. Because blockchain is an early technology smart contract languages, or even other dApps tools are constantly evolving. What today might be considered the most secure or decentralized tool or language, tomorrow might not, leading also to deprecated functionality.
- **Unit tests** don't cover all critical paths, and for so, bugs can be missed.
- **Human error**. Misinterpretation of test results or security analysis reports could lead to not noticing important issues.
- **Dependency issues**. Security vulnerabilities in libraries can lead to an impact on test validity and security analysis.

2.6 Analysis/Conclusions

Even though many solutions and scientific papers are presented, companies and individuals still face the problem of document counterfeiting/forgery, leading to them implementing, as a workaround, complex processes. Many solutions to this problem were provided, such as verification systems in broad and specific areas or even file-sharing and data storage solutions.

It was possible to see that no direct competitor exists and that the dissertation solution **cannot be compared** with any scientific paper since, that even though they try to solve document forgery, they address different users and for so different needs: some prefer decentralization over security and others prefer security over decentralization, this doesn't make one solution better than the other. Furthermore, some solutions are developed for a specific use case, such as medical or educational, and so they cannot be compared with each other or even with the one being developed (aims to be broadly used). It's also hard to compare the current solution in terms of quantitative values since the found articles don't provide enough information to make this comparison.

Furthermore, none of the papers provide an in-depth analysis of the challenges, pros, and cons, when developing in a public blockchain with the goal of maintaining a balance between security and decentralization.

Chapter 3

Technologies

This section gives an in-depth description and justification of the chosen technologies. To better illustrate the content of this section, a **Technology Diagram** is presented in appendix D.1.

3.1 Hybrid Networks vs Public Network

Because the solution has to support future developments, such as incorporating companies as end-users, it was considered to use either a hybrid or public network.

On the one hand, a **hybrid network** would enable a private network to coexist with a public one in one system. The private network could be used by companies - to manage their private files - and a public network could be used between individuals and companies, adding centralization only where it was needed. Nevertheless, this approach **doesn't show up as being suitable**. The overlapping of networks suggests that there would be a location storing the information of which network stored a given information. This would introduce a single point of failure, and for so, going against the main characteristic of this solution, which is decentralization.

On the other hand, a **public network**, although not as secure as private networks, promotes more decentralization, so it was the chosen approach. Encryption, permission, and logging mechanisms were implemented to enable document privacy and security.

3.2 Front-end Technologies

To make the dApp more accessible to a wide range of users, the solution was chosen as a **web app**. To this end, JavaScript and React were used as programming languages, npm as the package manager, Jest as the coverage testing tool, [48], and ESLint as the security analysis tool, [28].

Jest was the chosen testing tool for its ease of integration, community support, and built-in features. Furthermore, EsLint was the chosen tool for security coverage over the front-end code. This tool, even though not solely dedicated to security analysis, can be extended with plugins to cover security-related rules, such as "ESLint-plugin-xss". It detects 58 possible logic errors in the code, as well as provides suggestions for the errors encountered. This tool was chosen because of its ease of integration and free usage.

3.3 Back-end Technologies

This section compares different backend technologies/languages used in smart contracts and wallets, decentralized storage systems, frameworks, and tools for testing and security coverage. The chosen technology is presented for each of these, and a justification is given for why it was chosen.

3.3.0.1 Blockchain

The first technology decision was to choose a **blockchain** capable of responding to the solutions' needs. For this, some were taken into account:

- **Bitcoin Blockchain -** built to develop a store of value, not providing Turing complete smart contracts and not enabling the development of dApps. [57] [3]
- **Hyperledger Blockchain -** private, permissioned blockchain [2], it provides a second level of security by giving a node the power to decide who is eligible to join the network [91]. By adding this security measure, Hyperledger sacrifices decentralization, being the node, now considered a central point [34]. Furthermore, because Hyperledger doesn't use mining, each transaction performed is not paid. Because users' actions over a dApp are translated into transactions, by not paying for transactions, users don't have to pay for every interaction they make over the dApp [34] [135]. Nevertheless, despite these advantages, because the solution prioritizes decentralization and transparency, a private/permissioned blockchain, such as Hyperledger, doesn't fulfill the applications' needs.
- **Ethereum Blockchain -** public, permissionless blockchain. [135] It provides smart contracts that enable developing Decentralized Applications. Security is provided through smart contracts and decentralization, and there is no single point of failure.

Considering this analysis, **Ethereum** was the chosen one. The solution aims to be fully transparent and decentralized, allowing the sacrifice of security features that private, permissioned blockchains provide.

To work around these **security issues** and guarantee user confidentiality in their documents' content, data is always **encrypted** before being sent to the blockchain or IPFS.

Whereas Hyperledger used centralization to ensure that only wanted people could **access** a document, the system implements access **document permission** features. This way, it's possible to ensure document privacy without compromising decentralization.

Furthermore, to work around the **payment of every action** performed over the dApp, the less possible amount of information is sent to the blockchain, and IPFS stores the file content. Furthermore, a middleman could be implemented so that transactions could be cheaper to users. This mechanism is further explained in section 8.1.

It is noticed that Hyperledger can handle 1000 **transactions per second**, whereas Ethereum handles 15 transactions/second. Nevertheless, because the current dApp is a file manager and it's not a fundamental requirement for transactions to be executed rapidly, it was decided to abdicate from fast transactions to get a more decentralized application. [88] [2]

Lastly, it was seen that, by changing from PoW to PoS, Ethereum became less vulnerable to attacks like 51% attack or Sybill attacks since the malicious nodes incur high costs. Nevertheless, this blockchain remains susceptible to attacks such as 'Long-Range Attacks,' 'Denial-of-Services Attacks (DoS),' and Smart Contract Vulnerabilities. Since the **dApp will inherit the blockchain vulnerabilities**, these should also be considered.

3.3.0.2 Smart Contracts

As for the smart contract programming language, in between Vyper, [123], and Solidity, [108], **Solidity** was chosen. Even though Vyper is a more secure language, it's still not fully developed, it doesn't have as much community support, and lacks programming characteristics like inheritance. Furthermore, Vyper also introduces its own security vulnerabilities that are not present in Solidity. [84] [50]

It is worth noticing that Solidity still presents different characteristics when compared to other languages. Although accepting object-oriented programming characteristics, this language doesn't directly enable static methods. Instead, similar behavior can be achieved using functions that do not depend on the smart contract state, such as 'view' or 'pure' functions.

- **View**. These are functions that can't modify the state of the blockchain but can read from it.
- **Pure**. These functions can't modify or read from the state of the blockchain.

Furthermore, similarly to other languages, Solidity provides access modifiers, a fundamental security feature. These are [109]:

- **Public:** function is accessible from outside and inside of the contract.
- **External:** functions can only be called externally, this is, from outside of the contract. For a method to be external, the method cannot modify the contract's state. A function that can only be called by other contracts.
- **Internal:** functions can only be called inside the contract or contracts derived from it.
- **Private:** functions can only be called inside the contract that declares them.

Lastly, Solidity provides keywords for error handling and for verification purposes [109]:

- **Revert():** is used to revert state changes and execution in case of error or invalid conditions. Commonly used to handle conditions where the input parameters are invalid or the execution encounters an unexpected situation. All the changes made to the state vary within the current transaction and are reverted, and any Ether sent with the transaction is refunded.
- **Require():** is used to validate conditions and enforce pre-conditions, post-conditions, and invariants within smart contracts.
- **Assert():** used to check for internal error and should indicate a bug in the contract if it evaluates to fake.

3.3.0.3 Wallets

For wallets, **Metamask** was the chosen one. Metamask provides a browser extension, not forcing users to install additional software. It provides extra security by enabling the user to manage multiple accounts with a single seed phrase, and it's open source and has good tools, documentation, and community support. [11] Metamask is also used to authenticate the user into the dApp.

3.3.0.4 Decentralized Storage System

Several decentralized storage systems exist, including IPFS, [43], Storj, [112], Filecoin, [29], Sia, [102], and Swarm, [117]. **IPFS** was chosen because of its content addressing and retrieval, decentralization, and integration with Web3 dApps.

- **Content Addressing and Retrieval**. IPFS uses CID, a unique value attributed to each file based on its content. This ensures file integrity and immutability. On the other hand, complexity is introduced by Storj, Filecoin, Sia, and Swarm since they use addressing models

based either on contracts or market-based mechanisms. This market-based mechanism means that the availability and cost of storage vary according to the supply and demand in the network. [85] [40] [136]

- **Decentralization**. IPFS provides a fully decentralized peer-to-peer network, ensuring no single point of failure exists. It efficiently distributes the content across nodes, improving latency and access speeds. On the other hand, Storj, Filecoin, Sia, and Swarm can introduce complexity to ensure constant availability and speed since they involve market-based storage rental models. This is, nodes in IPFS store content without financial compensation, beyond the incentives of a peer-to-peer network. [85] [40] [136]
- **Integration with Web3 dApps**. IPFS is designed to work seamlessly with other Web3 technologies. Storj, Filecoin, Sia, and Swarm require additional steps to integrate with web3 technologies, complicating the development process.

With this being said, IPFS consists of a web protocol that enables the storage of heavy data outside of the blockchain and in a decentralized, distributed, and secure way. It also allows the system to have a good amount of storage and to increase efficiency. [96]

IPFS **is** a network of nodes (computers) designed to **store** various types of content, including images, videos, applications, and websites. IPFS identifies files by their Content Identifier (**CID**). Once a file is requested, IPFS finds the closest node storing it and retrieves the requested document. This retrieving process is based on the content and not the location of the information. It is important to note that not every node in the network has to store files, but every single node is responsible for broadcasting the files' location.[82]

Files are **not permanently kept** in IPFS. To ensure the permanency of a file in the network, the file must be **pinned** to a node. If a file is stored temporarily or unpinned, it will eventually be removed, making it inaccessible unless another node has a copy. [82]

Because storage is being spent, this pinning process **has a cost**. Monthly fees are associated, and the value depends essentially on three main factors [81]:

- **Storage:** The more storage, the higher the price is. This storage is measured by the number of files and their size.
- **Bandwidth:** Each content stored is downloaded using an IPFS gateway made of data. The size of this data is the bandwidth,

- **Requests:** A small fee is charged whenever a user accesses the data from the network. This access is considered a request.

Besides pinning, it's also possible to ensure that a file is available by hosting personal nodes, enabling the user to determine for how long a file should exist. Nevertheless, maintaining nodes involves high computational power and a lot of developer efforts to manage the process.

An example of a pining solution is "**Pinata**," enquiring monthly costs. These can be better detailed in the links on the references: [83] [81].

A local IPFS node was used for the current solution to avoid costs. At first, the chosen platform was Infura, intending to use the IPFS API it provides. Nevertheless, the API was in maintenance when the solution was being developed, so this couldn't be the chosen approach.

Consequently, the **IPFS Desktop App** was downloaded, and a local node was created. [44]

3.3.0.5 Frameworks

As for Ethereum frameworks, Truffle, HardHat, and Brownie were the possible choices. Each offers a robust development environment featuring an integrated testing blockchain, which facilitates compiling, testing, and debugging smart contracts. [127]

Because Brownie uses Python (not the chosen language), and Truffle is currently getting deprecated, these were not chosen.

The chosen framework was **HardHat**, which supports solidity to write smart contracts and JavaScript to deploy the contracts. HardHat also provides support for smart contract testing and a local Ethereum blockchain to which smart contracts can be deployed. [127]

3.3.0.6 Testing and Security Analysis Tools

HardHat already has an integrated testing framework, so this one was the chosen tool to perform unit testing over smart contracts. This testing tool also allows to have code coverage analysis. Furthermore, **Ether-gas-reporter** was the chosen tool to provide information on the gas consumption of functions during test execution.

Lastly, **Slither** [105] enabled security coverage over the solution. This tool analyses code and outputs warnings, information, and problems the code might face. This tool was chosen for its ease of use, customization options, and informative reports. The red lines in the output represent big security flows that should be solved. The yellow lines can represent a vulnerability or not, depending on its context. Lastly, the green lines represent information.

3.4 Analysis/Conclusions

While developing the solution, **Ethereum** proved to be a challenging blockchain to use in the current use case.

- Because Ethereum is a **public blockchain**, anyone in the network can access its information. While this can be seen as an advantage because it promotes transparency, it can also be a challenge when private information, such as private keys, needs to be stored. To work around this problem, a solution similar to the one used by MetaMask was used, and a mnemonic was generated. This implementation can be better analyzed in the section 6.5.2. Once private keys were generated and securely stored in a decentralized manner, any other data that needed to be stored in the blockchain was first encrypted, enforcing security, confidentiality, and decentralization.

- Ethereum also implies **high costs when storing documents** in the blockchain. Nevertheless, only the needed metadata is stored in the blockchain. The file content is then stored in IPFS to enhance scalability.

- Every **action** in the dApp translates into a **transaction**, which, consequently, has a cost for the user. Nevertheless, to work around this problem, a middleman could be developed. More details can be found in section 8.1.

Even though it was possible to get around the previously listed problems, other solutions could be taken into account, such as using Solana or ICP public blockchains instead of Ethereum or using Polygon with Ethereum.

- **Internet Computer (ICP):** It has canisters, similar to smart contracts on Ethereum but with more flexibility in data storage concerns. These contain the code of the dApp and enable programmers to define what information will be publicly available in the blockchain (by declaring variables as public) and which variables will be privately available in the blockchain (by declaring variables as private). Furthermore, canisters also enable to give access permissions to that private information. This way, it's possible to design a system that contains sensitive data that is kept confidential and which can only be accessible by authorized users, solving the problem of private keys previously described. It is noticed that, even though data isn't publicly visible, ICP still uses consensus mechanisms to ensure consistency and reliability across network nodes, maintaining decentralization. While ICP is public, the internal state of the canisters isn't, ensuring privacy and decentralization. The ICP official documentation can be found in [41].

- **Solana:** Solana, like Ethereum, doesn't allow the storage of confidential information in the blockchain in a private way. Solana could solve the issue of high transaction costs that Ethereum has. It is a high-performance blockchain known for its speed and low transaction costs. [80]
- **Polygon and Ethereum:** Polygon works as a layer two that enables the integration of multiple different Ethereum blockchains. This way, each solution can choose the blockchain that best fits their requirements while still being able to communicate with other solutions. Polygon enhances interoperability, scalability, and flexibility. It also reduces transaction fees and enhances performance at a lower cost. Ethereum and Polygon represent a good combination of technologies. Ethereum provides a secure base layer, and Polygon provides a better solution adoption, allowing better interoperability, performance, and costs. Solves Ethereum scalability and cost problems.[113]

Furthermore, **Solidity** proved also to have its own complexity. Because it's such a new language, it doesn't provide built-in functions (ex: toLowerCase()), it doesn't allow iterating directly over maps (it is necessary to keep a separate list of the keys), it doesn't have the "null" value, keys in maps can't be structs, it cannot receive maps as arguments, and doesn't allow string comparison in a simple way. Nevertheless, it was still a good choice, even though it required a little bit more effort on the developer side.

MetaMask proved to be a good tool, but it also provided some challenges while developing. When an error was sent from the smart contracts, MetaMask caught those errors and didn't enable the front-end code to understand why that error was occurring. This problem would persist when trying to integrate into the Solidity code "require()" or "revert()" keywords, more details on them in section 3.3.0.2. Nevertheless, MetaMask enabled a good and easy connection to the user's wallet, fulfilling its main goal.

To use **IPFS**, at first, the chosen platform was Infura. Nevertheless, its IPFS API was being maintained and could not be used then. Consequently, the IPFS Desktop App was used, and a local node was created. This approach and IPFS were good choices for storing files' content outside the dApp.

Lastly, **security testing tools and coverage tools** proved to work as expected. Security tools had a good range of attacks that were susceptible to happen, taking into account code vulnerabilities.

With this being said, it is possible to conclude that blockchain is a newborn technology, and its associated tools are in constant evolution and early stages. Some challenges were faced while using these technologies, but all were able to be solved.

Chapter 4

Requirements and Functionalities

This chapter aims to completely describe the solution regarding goals, requirements, and use cases.

4.1 Solution

The dApp being developed is a decentralized file storage and manager system. It aims to avoid document forgery and enable users to prove their identity by using this solution.

Transparency, security, decentralization, and confidentiality are the system's main goals and can be achieved through the many features the system provides. The system enables the management of files: upload, download, edit, delete, and share, as well as the registration of the users without having to keep any of their personal information. Because the system uses blockchain as base technology, it inherits its decentralization, transparency, and security features.

Furthermore, for a more **transparent** solution, an audit log over the users' files is also a feature. This enables the users with access to a file to control every action performed over them. **Security** can be a consequence of the use of blockchain, but this is not enough. Blockchain still has vulnerabilities that need to be addressed. Therefore, the solution provides extra security by encrypting files, using mnemonic generation, storing the private key locally and never on the chain, hashing the mnemonic, and using system timeouts. Lastly, **confidentiality** is ensured by the file symmetric encryption and the share functionality based on permissions.

Throughout the system usage, files go through state changes. These changes are illustrated in a **state diagram** presented in appendix D.2. The life cycle of a file after upload can be described as follows: Initially, the file enters an active state. While in this state, actions such as verification, sharing, updating permissions, and downloading maintain the file's active status. However, if the file is edited, it transitions to the edit state. In this state, only downloads are

permitted. Furthermore, if a delete action is executed on an active file, the file moves to a deactivated state, rendering all actions on the file impossible.

The sections to come will further detail the system in terms of Goals, requirements, and use cases.

4.2 Goals

The system's **main goal** is to promote trust in documents in order to solve document forgery and counterfeiting. To achieve this, the document should be a decentralized file manager and identity provider.

For **effective decentralization**, the system must operate without intermediaries and improve efficiency in process verification. Blockchain is the ideal technology for this, offering decentralization, security, transparency, and data integrity.

In order to **promote security** in the system, using blockchain is not enough, and other mechanisms, such as encryption, are used to ensure users' data protection and good compliance with data protection regulations. Furthermore, transparency is maintained by implementing audit logs that track users' activities and file-related events, with this last one also reinforcing file integrity. Moreover, the presence of sharing and verification mechanisms further strengthens transparency.

Lastly, because the system should also work as a **document manager**, functionalities such as document storage, editing, uploading, and sharing should also be considered system goals.

A **goal tree** was used to visually represent these goals. This diagram is presented in appendix D.3, where each goal in this tree represents the desired outcome that the project aims to achieve.

Many of these goals were **identified** by reviewing the project's description provided by the company NearSoft, during the meetings held throughout the semester and by analyzing similar existing systems.

4.3 Requirements

The dApp requirements are divided into **business requirements**, **blockchain requirements**, **non-functional requirements**, and **functional requirements**. These requirements were elicited from meetings held throughout the semester, from the information on the project description in Sigarra, and from research of existing software in the market and scientific papers.

The tables listing these requirements are in the appendix C for better document organization. As it is possible to see, a MoSCow method was used, where:

- **M: Must-have**. For requirements that are needed for the project to be completed.

- **S: Should-have**. For requirements that are important but not critical for the system to work.
- **C: Could-have**. For requirements corresponding to desirable features that don't affect the projects' success.
- **W: Won't-have**. This is for requirements with the lowest priority or that are not necessary for the timeline.

4.4 Use Cases

The system only has one user that can represent a single entity or a complex entity such as a company. To easily visualize the use cases, a **use case diagram** is presented in D.4. It is possible to see that the system was also considered an actor since it also performs functionality instead of the user doing it.

4.5 Systems Limitations and Dependencies

The system is **dependent** on **IPFS**, **MetaMask**, and the **Ethereum Blockchain**. If any of these applications stop working, suffer an attack, or change anything in how they work, the current solution will also be affected.

As for the system **limitations**, the solution has only been developed to interact with the **Ethereum** blockchain and **MetaMask** wallet. Furthermore, login is only possible through a **mnemonic** generated in the users' registration. As such, if the user loses his mnemonic, he loses access to his account forever, and if someone else gains access to it, the user's account might get compromised. Furthermore, every **user's action** corresponds to a transaction, and as a consequence, every user's action has a price, which might hinder users from adhering to the dApp. As compensation for this last limitation, IPFS is used to decrease the transaction cost, and a middleman can be implemented (explained in section 8.1).

4.6 Analysis/Conclusions

Looking at the goal tree, all goals but "Use this app as an identity provider for other apps," "Manage user identity," and "Implement an audit log for users activities" were implemented. The first two unimplemented goals were not fully achieved since the user can not use the app to access other apps. Nevertheless, it is capable of reliably and securely storing documents that guarantee users' identity, such as identity cards, driving licenses, and others.

While developing the dApp, the priority column in tables of appendix C was considered.

In terms of **business requirements**, all were implemented. The dependence on intermediaries was reduced by using blockchain technology alongside Hierarchical Deterministic Wallet (1.5). Furthermore, using IPFS decreased file storage costs, as only essential information was stored on the blockchain. All file management features were fully implemented, thus enabling secure user verification and file access. To further reduce costs for the user, the introduction of a middleman could be considered, as detailed in section 8.1.

All **blockchain requirements** were also achieved. Besides the chosen blockchain, Ethereum, being considered public and permissionless, only the needed information is stored in it: the files' metadata, the user information, and the permissions the users have over the files.

Similarly, all **non-functional requirements** were also implemented. Blockchain itself leverages features like authenticity, integrity, security, and decentralization. Additionally, the audit log ensures transparency, while file verification ensures document authenticity. Because data is encrypted before being stored in the blockchain, so no confidential data is set public, adhering to data protection regulations.

Unlike other requirements, not all **functional requirements** could be implemented. The Audit Log only records actions performed on files, not user actions. As a result, user actions during file validation appear in the audit log only if the validation is successful, i.e., if there is a match with a file stored on the blockchain. The functionality of sharing files via a link and Audit Log filters was not implemented. Furthermore, the application does not yet support using this app to log into other apps. Nevertheless, it is worth noticing that all "Must-have" requirements were implemented, indicating overall success in achieving the desired features.

Throughout the development, continuous meetings and market solutions analyses were performed to refine requirements. As such, requirements evolved over time alongside their implementation.

Chapter 5

Architecture

This chapter provides an abstract view of the system by describing its architecture and providing a component diagram for better visualization. Section D.5.1 illustrates an overview of the system and how the different components interact.

5.1 Component Diagram

To represent the components of the system, a **component diagram** is used (appendix D.5.2). There are 2 main components: NearFileWebApp and NearFileBackend.

NearFileWebApp is the component that interfaces the system being developed. While the user interacts with the application through React Components, MetaMask, and IPFS interact with the application using the Helper component. While the user is the one who intends to use the system and its functionalities, Metamask is the tool that facilitates wallet connections and transactions, and IPFS is the tool responsible for storing and retrieving the file's content. This component contains three more others besides the React Components:

- **React Components**. This is the component that makes up the user interface of the dApp. It has React classes and enables the user to have a friendly way to execute the features the dApp provides.
- **EthContext**. The context provider manages Ethereum-related states like user accounts, connections, and smart contract instances.
- **Helpers**. It contains logic that supports the interaction between the front end and the smart contracts.

- **Contracts ABI**. It contains the smart contracts' **ABI**, which contains the smart contracts' structure and establishes how the front end communicates with the back end.

It is the Contracts ABI and the EthContext components that enable the communication with the back end. This communication is performed through **contract calls/transactions**. While smart contract **calls** are read-only interactions, smart contract **transactions** are state-changing interactions(they change the blockchains' state). This second type of interaction requires gas consumption.

NearFileBackend, on the other hand, interacts with the Ethereum blockchain, where smart contracts are deployed to and executed. This component has the following sub-components:

- **Contracts**. Contains smart contracts written in Solidity. These contracts handle on-chain login and data storage. They are responsible for running the function requested by the front end and returning data (if a call is requested) or sending confirmation (if a transaction is requested).
- **Deployment Script**. Used to deploy smart contracts into the Ethereum network.
- **Tests**. It contains code that is responsible for testing the solidity smart contracts. It communicates with CoinMarketCap to enable **eth-gas-reporter** to understand how much a transaction costs while executing the tests.

5.2 Analysis/Conclusions

Some architectural challenges were found when developing the solution. HardHat was being used as a back-end framework. When deploying smart contracts, it was noticed the creation of a contracts folder in the front-end component. These folders contain the smart contracts' ABI. Nevertheless, they did not contain the address to which these smart contracts were deployed. As a solution, when writing the deployment script, it was manually programmed to create files storing the smart contracts' addresses. This way, these can be used and called by the front end.

The described architecture presents a **good separation of concerns**, **scalability**, and an **interactive user interface**. The back end and the front end, as well as their sub-components, are clearly separated, enabling greater maintainability, clarity, and debugging. Because React enables code reuse and efficient rendering, and smart contracts provide a load distribution through decentralization, scalability is achieved both in the front and back end. MetaMask also integrates greater wallet security. Furthermore, blockchain interactions are hidden from the user, who only interacts with the app using the react component, guaranteeing a greater user experience.

Nonetheless, the architecture also poses some challenges. The system involves multiple technologies and components, and consequently, maintaining it might require developers to be familiar with tools and knowledge in various domains, such as IPFS, MetaMask, and Ethereum blockchain. Furthermore, performance bottlenecks might also be felt because of gas costs and blockchain latency. This challenge can be solved by implementing a middleman, explained in section 8.1. The system is vulnerable to smart contract attacks, as it also depends on issues that might happen in the Ethereum Blockchain, MetaMask, or IPFS. Lastly, using MetaMask can pose a barrier for users to use the app since this extension must be installed in the browser, and the user might not be familiar with this blockchain technology.

Chapter 6

Implementation

This chapter aims to detail how the development of the solution was performed, beginning with an explanation of data management, illustrated through the **entity-relationship diagram**. Furthermore, all classes of the solution and the relationship between them are further explored using a **class diagram**. Lastly, a detailed explanation of feature implementation is provided, followed by **flowcharts**.

6.1 Entity Relationship Diagram

The Entity Relationship Diagram is presented in appendix D.6. In this diagram, primary keys are represented in bolt, and foreign keys are underlined. Further than this, it is possible to see **five entities:**

- **File**. Contains the files' attributes. The smart contract for managing this data is "FileRegister.sol".
- **User**. Contains the the users' attributes. The smart contract for managing this data is "UserRegister.sol".
- **FileHasUser**. It contains the information needed to connect a user with a file. The smart contract for managing this data is "AccessControl.sol".
- **Audit Log**. Contains the audit log attributes. The smart contract for managing this data is "AuditLogControl.sol".
- **Login State**. Contains the information related to the users' login. The smart contract for managing this data is "LoginRegister.sol".

Through the analysis of this diagram, it's possible to see a **time-out mechanism** implemented by "LoginRegister.sol", which is further explained in sections 1.8 and 6.5.1. Because each user has only one state of whether he is logged in, there is a **relationship** with the user of one-to-one. Furthermore, because the **users** can upload and share **files**, a file can belong to multiple users, and multiple users can access the same file. Therefore, a relationship of many-to-many is established. Because the **type of access** can be provided when a file is being shared, this property belongs to the relationship "user file". Therefore, this attribute is in the table "FileHasUser." Furthermore, each file is encrypted using a **symmetric key**, which is then encrypted using the user's public key. Therefore, this property depends on data stored in the file and user tables. This encryption mechanism is better detailed in section 1.3 and 6.5.4. Lastly, because the **audit log** shows every action executed over a **file**, this table establishes a one-to-many relationship with the "FileHasUser" table. This way, the audit log can track which files are accessible to which users.

6.2 System Interfaces

A flowchart on how interfaces are related and the logic behind it is presented in appendix D.7. There are three main pages: "Connect Wallet", "Login", "Register", and the "Main Page". It's also worth noticing there is a **timeout mechanism** implemented. This mechanism is better detailed in sections 1.8 and 6.5.1.

6.3 Front end Class Diagram

To understand how classes are related and the attributes and methods they possess, a **Class Diagram** is presented in appendix D.8. The **naming** of each class is associated with their role in the design pattern they are part of. Note that "BlockchainWrapper", "IpfsWrapper", and "EncryptionWrapper" are part of the complex system being hidden from the clients. "Wrapper" was the name given since it encapsulates functionality related to a specific area. Three design patterns were implemented: **Facade Pattern**, **Command Pattern**, and **Template Method**.

Facade Pattern was implemented to abstract clients (React Components) from the system's logic. Here, clients only communicate with the "FileManagerFacade.js" which knows which classes must be called to execute the given functionality. This way, **loose coupling** between the interface design code and the logic code is ensured. Three **wrappers** were developed to **organize** better the complex subsystem: Blockchain, IPFS, and Encryption Wrappers. Each of these encapsulates functionality related to a specific area of concern. This structure obeys the **Single**

Responsibility Principle and enables an **easy refactor**, if something changes in the smart contract code, the only place to change is the Blockchain Wrapper. This pattern also gives space for system **evolution**. For example, if another blockchain is integrated into the solution and the user can choose which blockchain he wants to interact with, a **Decorator Pattern** could start to be developed in the BlockchainWrapper.

Command pattern was implemented to organize the multiple actions possible to be executed over a file. This pattern goes along with the **Single Responsibility Principle** since it encapsulates the logic of actions, such as delete, upload, or edit, in separate classes. **Loose coupling** is also applied between the "FileManagerFacade" and the commands by passing the needed methods as arguments instead of the FileManagerFacade instance itself. This way, if the instance changes, commands do not get affected.

Lastly, **Template Method** was implemented to avoid code repetition. DropFile and DropUpload classes share the same code structure but change only how the file is stored. Therefore, the "storeFile" method is the **hook method** that enables each subclass to implement accordingly to their needs.

6.4 Back end Class Diagram

To understand how classes are related, their attributes and methods, a **Class Diagram** is presented in appendix D.9.

While the "**AccessControl**" contract is responsible for managing the "**UserRegister**" and the "**FileRegister**", these both are responsible for managing specific data related to the user and the file, respectively. Moreover, both "**AuditLogControl**" and "**LoginRegister**" are responsible for managing audit logs and login mechanisms, respectively. Lastly, "**Helper**" is responsible for storing methods that are independent of the contracts' state/logic.

Each smart contract reflected in the diagram is **responsible** for managing and maintaining the attributes they possess. To guarantee consistency throughout the development and a better code understanding, it was established that all methods that return information would do it in a standardized way, defined by **structs**. These structs would have a boolean success variable, a result variable, and a message variable. This last variable contains information in case something goes unexpected. This is done to avoid returning an empty object, which is not a good practice, and because solidity doesn't support the keyword "null".

Furthermore, the "**AccessControl**" contract is responsible for deploying other smart contracts. This is done to avoid **cyclic dependency**. Because some methods, for example, in the "FileRegister" can only be called by the "AccessControl" smart contract, and because the "AccessControl" smart contract calls methods in the "FileRegister", when deploying these contracts, a cyclic dependency was found. This deployment process can be found in appendix E.5.

6.5 Features

To understand how the solution's features were implemented, this section details each and accompanies them with flowcharts in appendix D.

6.5.1 Timeout

Implementing a timeout mechanism on the front end is not enough. Because smart contracts are deployed to the Ethereum blockchain, its code becomes accessible to everyone in the network to use and exploit. Therefore, this should also have its own timeout mechanism. This section explains the chosen approaches, and the flowcharts can be found in appendix D.10 and D.11.

In the **front end**, this logic is implemented in the class "SessionExpirationHandler. js". This class tracks all mouse movements. Every second, it checks whether the mouse has been inactive for a certain period. If the inactivity exceeds a defined threshold, the user is logged out. Furthermore, if a page refresh is performed, the count is reset.

In the **back end**, this logic is implemented in the "LoginRegister.sol" smart contract. When a user is registering, besides adding the user to the blockchain, a "logged" variable is set to true, and the time of the registration is also set. The same approach is used when the user logs in, the "logged" variable is set to true, and the log-in time is also set. More details about how login and registration are implemented can be seen in section 6.5.2.

Because the variable "logged" stores the user's state in the dApp, every time a user executes a transaction, the transaction verifies if the user is logged in, and if so, the transaction gets executed.

The user gets logged out if he logs out from the application or if the time the user is in the dApp has reached a threshold. If the user logs out for these reasons, the "logged" variable is set to false. The time the user can execute transactions is hard coded in the smart contract; this

helps prevent a hacker from changing the timeout value.

Unlike what happens in the front end, the back end doesn't consider the user's last action. In the back end, there is a time limit between login and logout, which allows the user to execute transactions that have not been updated. On the other hand, there is a time limit for users' inactivity on the front end. If the user moves the mouse, the time is updated. The ideal would be for the back end to perform the same implementation as the front end. This is a future improvement, as indicated in section 8.

6.5.2 Registration and Log in

The flowchart for this feature is in appendix D.12, and a code snippet is in appendix E.4. Similarly to MetaMask, this solution's registration and login process is based on the concept of Hierarchical Deterministic (HD) Wallets. More detail on HD wallets can be found in section 1.5.

The **Registration** is performed when the user's account is not stored in the system, creating a **mnemonic** that the user should keep secret and safe. An **account** with a **public and private key** is associated with this mnemonic. While the mnemonic is **hashed** in the blockchain alongside the generated public key, the private key is stored in the user's **local storage**. This way, security is added by storing the private key locally, never leaving the users' device, and hashing the mnemonic (1.4), making it impossible to get its value back.

The **Login** is performed when the user's account is already in the system, leading the user to insert the mnemonic provided in the registration. The inserted mnemonic is hashed and compared to the value stored in the blockchain. If these two are equal, the private and public keys are regenerated. The private key is kept in local storage, and the public key is in the blockchain. The user gets redirected to the main page. By providing the correct mnemonic, the user can prove its identity.

This way, it is important for the user to maintain the mnemonic secret and never forget since it is responsible for getting the private key back (when a login is asked again). The mnemonic can be seen as a password that the user utilizes when he needs to get access to his account.

Once the user logs in or registers, the system initializes the smart contracts and other needed variables, as shown in the flowchart in appendix D.13. In the code snippet, we use the HD to generate an account and use its private and public keys.

6.5.3 Log out

The flowchart for this feature is in appendix D.14. It's possible to log out by either exceeding the **session time** limit or clicking the logout button. Once this is done, the log state of the user is set to false, and all the contract initialization, alongside other variables, is reset.

6.5.4 Upload File

The flowchart for this feature can be seen in appendix D.15. When a user uploads a file, the file is encrypted before being stored in IPFS. Because **symmetric encryption** is suitable for encrypting large amounts of data, this was the chosen approach to encrypt files. More details on symmetric encryption are in section 1.3.

Appendix F shows a request to the local IPFS node containing a stored file. It's possible to see the file is effectively encrypted since a bunch of random characters are displayed. Before using Postman, the local IPFS node was used to perform this request, but the app did not recognize these characters, so Postman was the chosen tool. When IPFS stores a file, the corresponding CID is returned. This parameter is unique for each file and allows a way to get the file back.

For a user to be able to decrypt the encrypted file stored in IPFS, the user should have the symmetric key associated with the file. For this reason, there is a relation between a file and a symmetric key, which is then stored in the blockchain. Because it's possible to see everything stored in it, encrypting the symmetric key first is important, preventing unauthorized people from accessing the file.

To this end, **asymmetric encryption** was chosen, more detail in asymmetric encryption in section 1.3. The symmetric key is encrypted using the public key of the users who have access to the file. Afterward, when the user wants to see the file's content again, he only needs to use his private key to decrypt the symmetric key and then use the symmetric key to decrypt the file. This way, there's only one symmetric key for every file but multiple encrypted symmetric keys - resulting from the encryption with the users' public key. This ensures that only users with access to the file can see the symmetric key and, therefore, access the file.

The CID, the encrypted symmetric key, the file version (set to 0), and the file hash (used in verification) are stored in the blockchain. If the user is logged in, the action gets executed in the blockchain and maintained in the audit log.

6.5.5 Edit File

The flowchart for this feature can be seen in appendix D.16. This action is similar to the Upload File. When a file is dropped, a new symmetric key for the current file is generated and

used to encrypt it before storing it in IPFS. This symmetric key gets encrypted by the public key of the user accessing the edited file.

Before storing this new editing in the blockchain, the version gets incremented, the file owner is kept as the user of the file with version 0, and the state of the previous file is set to "edited", while the current one is set to "active".

The new file is added to the blockchain if the user is logged in. Furthermore, every user who had access to the previous file now has access to the new edited file. Lastly, this edit action is stored in the audit log.

6.5.6 Download File

The flowchart for this feature is provided in appendix D.17. Once a file is selected, the system checks whether the user has download permissions. If permitted, the file is retrieved from IPFS using its IPFS CID, which is stored on the blockchain. Next, the user's private key, kept in local storage, decrypts the corresponding symmetric key encrypted on the blockchain. This decrypted symmetric key is then used to decrypt the file, enabling it to be downloaded.

6.5.7 Delete File

The flowchart for this feature can be seen in appendix D.18. Once the user has selected a file, if he has deleted permissions over it and the session time has not expired, the file state is set to deactivate, and the action gets stored in the audit log. Nevertheless, when a file is deleted, all its past editing is kept in the edited state. These file state transformations can be seen in the **State Diagram**, appendix D.2. Because the audit log maintains even after a file has been deleted, the relationship between a user and the file is kept after a file deletion. Lastly, the file is not deleted from IPFS because the garbage collector does it automatically; for more detail, see section 3.3.0.4.

6.5.8 Share File / Update Permissions

The flowchart for this feature can be seen in appendix D.19. When the user performs the share of a file, he is asked to input a user's name and give the permissions the user will have over the respective file. If the user to share the file is not already associated with the file, the system considers this action as being a file **sharing**. But, if the user is already associated, the system considers it a file **update permissions**. To **remove the access** of a user over a file, all permissions of the respective user must be removed. For a better organization, the next titles separately describe the case where the user performs a file share and the case where the user performs a file update permissions.

6.5.8.1 Share File

When user A shares a file with user B, the dApp grabs from the blockchain the encrypted symmetric key between user A and the file, alongside the encrypted symmetric key of past file edits. Then, the decryption of these symmetric keys is performed using the private key stored in the local storage. Afterward, the system grabs, from the blockchain, the public key of user B and encrypts the decrypted symmetric keys with it. This way, user B starts accessing the symmetric key responsible for decrypting the current shared file and all its past edits. This relationship is then stored in the blockchain, and the action is stored in the audit log.

6.5.8.2 Update Permissions

If user B is already associated with a file that user A is trying to share, if user A provided user B with permissions, then this field gets updated; otherwise, if all permissions were removed, the association of user B with the file gets deleted. This action is also stored in the audit log.

6.5.9 Verify File

The flowchart for this feature can be seen in appendix D.20. When the user uploads or edits a file, the file hash is first calculated and stored in the blockchain as file metadata. Then, when a verification is requested over a dropped file, the hash is calculated and compared with the hash of the files in the active state already in the blockchain. If a file with the same hash is found, the dropped file is considered valid and otherwise invalid. If the dropped file has the same hash as a deleted or edited file, state "delete" and "edited" respectively, the dropped file is not considered valid.

File hashing was chosen because, once using a strong algorithm like SHA-256 for the same content, the same hash is always generated, and the hash of the file is unique to its content; this is, it's not possible to have two different contents with the same hash. More information on this can be seen in section 1.4.

6.5.10 File Versioning

The flowchart for this feature can be seen in appendix D.16. File versioning is performed when the user uploads and edits a file. By increasing the version in each edit and changing the file states to "edited" (old file) and "active" (new file), and by keeping the owner, the user who uploaded the file in version 0.

6.5.11 Audit Log

If a user has access to a file, they can see past actions over it from the upload, edit, and download to its share. The audit log is only performed over the files, not the users' actions. Consequently, when a file verification is performed, it only shows in the audit log if the verification was successful and if it matched with another file in the active state. On the other hand, if the match was unsuccessful, it didn't match with no file, so it's impossible to audit the log. If permissions are taken out from the user, the user stops from having access to the audit log of the file.

6.6 Security

The described solution is developed over the blockchain, which means that even though it inherits its security features (section 1.1), it also inherits its vulnerabilities.

Various security measures had to be considered to answer to these vulnerabilities. This section aims to describe them by dividing the security concerns into topics. This can also be supported by section 1.2.

- **Blockchain selection**. Since the dApp inherits the blockchains' characteristics, ensuring the chosen one has the needed security features is important. For this reason, the various blockchains were analyzed in section 3.3.0.1. Ethereum blockchain mitigates many security vulnerabilities by applying the PoS consensus mechanism and penalties to those with malicious behaviors. For this reason, attacks like '51% Attack' or 'Sybil Attack' are less prone to happen in this blockchain. Nevertheless, this blockchain is still prone to attacks, especially in smart contracts. For this reason, other security measures had to be taken.

- **Timeout mechanisms**. Section 1.8 details the vulnerabilities that can happen without a session timeout. The current dApp implements this mechanism on both the front and back end to avoid those vulnerabilities. The chosen approach in each of these is in section 6.5.1. In the backend, the time to which the user is going to be logged out is defined in a hardcoded manner, which helps prevent hackers from being able to change the timeout value.

- **Data Confidentiality and Integrity**. The current dApp must present great confidentiality and integrity on the stored files. To this end, various encryption and hashing mechanisms were performed. To ensure **documents' confidentiality**, these are encrypted using symmetric encryption with a strong algorithm: **AES-256** with the **CBC mode**. This was the chosen algorithm since AES is recognized as robust, and CBC mode is able to ensure that

identical plaintext blocks produce different ciphertext blocks, making it harder for attackers to deduce patterns and infer information from the ciphertext. Symmetric encryption uses a **symmetric key** that is then distributed to all users with access to the file. This way, only the users with permission can get the files' content back. Because it's also important to ensure the **symmetric key confidentiality**, this one had to be encrypted using **asymmetric encryption**. With this mechanism, the symmetric key is encrypted using the respective users' public key and then stored in the blockchain. Afterward, if the user wants to get the files' content back, they have to decrypt the symmetric key using their private key and use the decrypted symmetric key to decrypt the file. This way, it's possible to safely store the file in IPFS and the symmetric key in the blockchain. Lastly, to ensure the users' **private key confidentiality**, a Hierarchical Deterministic Wallet (HDW) was developed. This process involves generating the same public and private key for the same mnemonic using Ethereum's Elliptic Curve Cryptography (ECDSA) with a 256-bit key size (specific elliptic curve used by Ethereum). This process is better analyzed in section 1.5. Once the private key is generated, it will be stored in the user's local storage (browser), never leaving his device.

This mnemonic is generated when the user performs a registration. To ensure **mnemonic integrity and confidentiality**, the mnemonic gets first hashed and then stored in the blockchain (1.4). Once it gets hashed, it's almost impossible to get its value back. Nevertheless, the algorithm used for hashing is SHA-256, which is a fast hash function, and for so, it's less secure when compared to slow hash functions, more detailed in section 8.1. On the next login attempts, the user inserts the given mnemonic; his input is hashed with the same algorithm and compared with the respective one stored in the blockchain. If these match, the private and public keys are generated based on the users' input. This way, it's possible to ensure both integrity and confidentiality of the mnemonic.

Furthermore, to ensure the **files' integrity**, they are hashed and stored in the blockchain. Later, when the user tries to verify the validity of a file, the inputted file gets hashed and compared with the ones in the blockchain (6.5.9). Furthermore, file integrity is also ensured by storing them in IPFS. IPFS retrieves a file based on its content and not based on its location.

File confidentiality is also ensured by attributing permissions over a file. Extra security is added to the file and symmetric key by always encrypting them on the client side, never leaving their device.

Lastly, as explained in section 1.1.3, any program that compares hashes is vulnerable to "Hash Collision Attack". This, in the case of the current solution, could lead to false

positives in case of file verification or mnemonics to match even though they are different. Nevertheless, to avoid this attack, strong encryption algorithms were used.

These measures prevent eavesdropping and unauthorized access.

- **Input Validation**. Input validation is important both in the front end and back end.

  Front-end input validation is important to avoid Cross-Site Scripting (XSS). This attack is about injecting malicious scripts into web pages that other users can access. To prevent this attack, inputs are always validated to ensure they assume the expected structure. An example is the input name in the registration, where the user's name should be a lowercase unique string.

  This security measure gains another level of importance in the back end since smart contracts, once deployed, are visible to everyone in the network. For this reason, input validation is important to ensure the received arguments assume the expected value, avoiding possible attack vectors. One common attack is the "Short Address" attack, better detailed in section 1.1.3. To this end, when a function receives an address as an argument, the argument is validated if it is a valid address. An example is provided in section E.7.

- **Secure Authentication and Authorization**. **Authentication** is performed both on the back end and front end. In the back end, this is performed through a variable set to true once the user logs in and false once the user logs out. In the front end, this mechanism is ensured through a mnemonic. This mnemonic is securely generated through a mechanism described in 1.5.

  Furthermore, in authentication, a timeout mechanism is also implemented and explained in section 6.5.1. This gives an interval to which the user can interact with the dApp, ensuring less time for a bad-intentioned user to execute an attack vector.

  In the **authorization**, this mechanism is implemented both on the back end and front end. Each method in the back end has a set of validations executed before the action itself. These validations ensure the user has the authority to execute the procedure. An example can be seen in appendix E.1. Nevertheless, some of these methods don't need validation since they don't return any relevant information. An example of this can be seen in appendix E.2. In the back end, another feature that enhances secure authorization is the use of access modifiers both in functions and variables. These define who has access to execute a given function. These access modifiers can be further analyzed in section 3.3.0.2. Using them makes it possible to avoid variables and functions open to abuse and misuse. Lastly, the use of a Share functionality also provides a secure authorization, since it uses private and public keys to manage who has access to a file and who doesn't.

- **Smart Contract Robustness**. Smart contracts, once deployed to the blockchain, become visible and accessible to everyone. For this reason, it's important to ensure their correct functioning before deploying them. To this end, unit testing and security tools were integrated into the system. This helps to understand if the code is working as expected and if any code is susceptible to vulnerability.

- **Keeping up with the latest technology**. Efforts have been made throughout the development to keep the most updated technologies. At first, Truffle and Ganache were being used, but it was possible to understand they were being sunset. For this reason, the HardHat framework was chosen since it was being maintained. Another example is using the most recent solidity version, which avoids attacks like integer overflow and underflow. These attacks are further explained in section 1.1.3. The first security analysis tool for the back end was MythX, but then it was noticed that this tool was being sunset, and for so, it was substituted by Slither.

- **User Experience**. By implementing an interface, the user doesn't communicate directly with the blockchain, abstracting him from all the communications needed. Furthermore, validations are performed before calling a blockchain transaction to prevent the user from losing gas because of wrong inputs or because the conditions are not met. This can be seen in section E.10. Furthermore, information is provided to the user to educate him. An example is the warning when the mnemonic is given to him. This promotes awareness and prevents **phishing attacks**.

6.7 Analysis/Conclusion

An analysis of the **Entity-Relationship Diagram** reveals certain downfalls in the implementation. Notably, the attribute "encSymmetricKey" is linked to users regardless of their download permissions, suggesting a potential design flaw. Ideally, users should only be associated with an encrypted symmetric key if they have download permissions for a file. However, in cases where a file is shared with a user who has only delete permissions, they still receive an encrypted symmetric key despite being unable to perform downloads. Similarly, if download permissions are revoked from a user, the associated encrypted symmetric key should also be removed, but this action is not currently being taken. One possible solution could be only to provide the user with an encrypted symmetric key if he has download permissions and delete the key when the permissions are removed. Nevertheless, this would originate registers with empty values in the encrypted symmetric key, being an indication of a bad design. Another **solution** could be to have a table for each permission. This way, there would be a "deletePermission" table with the

attribute "encrypted symmetric key". When the user is given download permissions, a new register is added, and when the permission is removed, the register is deleted.

An analysis of the **Design Patterns** also reveals some challenges to the current solution. "**FileManagerFacade**" interfaces the entire system to the clients. Consequently, this class has an excessive number of methods, which only increases as new functionality is added to the system. Similarly, the "**BlockchainWrapper**" class also has excessive methods and tends to expand in complexity as new smart contracts and new methods on the back end are added. With this being analyzed, it becomes evident that even though the role of "FileManagerFacade" is to abstract the complexity of the system, and the role of the "BlockchainWrapper" is to centralize the blockchain functionalities, they inadvertently accumulate excessive responsibilities which can later complicate the systems' maintenance. A potential **solution** to the size of the FileManagerFacade could be to create three other facades: "BlockchainFacade", "IPFSFacade", and "EnncryptionFacade". Each of these would encapsulate the corresponding methods currently handled by the FileManagerFacade. This architectural adjustment would enable the FileManagerFacade to re-direct requests to the appropriate sub-facade based on the operation's nature. These sub-facades would then interact with the specific methods in their respective wrappers, ensuring a more organized and manageable distribution of responsibilities. Similarly, another potential **solution** to the size of the BlockchainWrapper could be to divide it into sub-wrappers according to the smart contracts.

With this implementation of wrappers, it's possible to encapsulate related operations, promote the single responsibility principle, and enable easier refactoring. Furthermore, the Facade Pattern also promotes decoupling.

It was decided to pass the methods as arguments in the **constructors** of the commands as a way to enable loose coupling. Nevertheless, the constructors started getting bigger and will tend to get even bigger as new functionality is added.

An analysis of the **Smart Contracts** also reveals certain downfalls in the implementation. The "AccessControl" smart contract became overly large and had too much responsibility, tending to worsen as new functionality was added. At a certain point, the smart contract deployment was no longer possible due to its size. To address this issue, the parameter "allowUnlimitedContractSize" was introduced in the "hardhat.config.js", a modification detailed in the code snippet found in appendix E.3. To **mitigate** this problem, the "Helper.sol" contract was developed. This new contract contains methods that do not directly require the attributes of "AccessControl.sol", thus reducing the burden on the primary smart contract and helping to manage its complexity

more effectively.

Furthermore, some functionalities are hard coded in the smart contracts, ensuring no bad-intentioned user can alter them. An example is how permissions are hardcoded in the method "uploadFile()" in the "AccessControl.sol" smart contract (appendix E.6). The disadvantage is that if the permissions are altered, the smart contract would have to be redeployed.

Challenges related to the **transaction costs** were also faced. Some user's actions imply the execution of multiple transactions. For instance, when a user uploads a file, the process involves multiple transactions: one to store the file on the blockchain and another to record the action in the audit log. This approach adheres to the "Single Responsibility Principle" by keeping separate concerns distinct. However, this method results in the user incurring double charges for what appears to be a single action—a scenario that is neither user-friendly nor cost-effective. To **solve** this problem, the "**Single Responsibility Principle**" **had to be put aside**. By integrating both tasks into a single method, the file is stored on the blockchain, and the action is logged simultaneously, all within one transaction (detailed in appendix E.1). Having a method with these two responsibilities violates the principle. Nevertheless, it was necessary to ensure a better user experience. A significant advantage of this approach is that the audit log is updated directly within the method once an action is executed, ensuring that only actual transactions are recorded on the blockchain. Additionally, restricting the ability to call this method to the "AccessControl" smart contract increases security by making it more difficult for unauthorized entities to manipulate the audit log or log fictitious actions.

Further into the **transaction costs**, even though their number is decreased to the minimum possible, they still have high costs to the end user. A solution for this would be to implement a middleman between the front end and the back end. This way, the front end sends transactions to the middleman, accumulating them, and once they reach a certain threshold, they are then executed in the blockchain. This way, transactions get bundled, and the costs are divided among the end users. Nevertheless, this was not possible to implement, being something to implement in the future as described in section 8.1.

Lastly, in the **transaction costs**, to ensure the user doesn't have costs over a transaction that fails due to programming errors, validations are performed to ensure the user is eligible to execute the transaction.

For a more **efficient solution**, when possible, **maps** were the data structures chosen to store

the systems' data since they allow for more efficient searching. Only the "AccessControl" smart contract has a list instead of a file since the key on the map would have to be a struct, and Solidity doesn't allow keys to be structs. The keys in these maps are what would be considered the primary keys in an Entity-Relationship Diagram.

The app doesn't follow the "**principle of least privilege**," which says users should only have access to the resources and information necessary to fulfill their roles. On the one hand, access modifiers in each function and variable declaration ensure the principle is being followed since users only access the methods and functions they are allowed to. On the other hand, as previously explained, users have the symmetric key of a file even though they don't have download permissions over it. The user has more information than the role he performs.

It's also worth noticing the classes don't prevent the **timestamp vulnerability** (detailed in section 1.1.3) as it is possible to see in appendix E.8. And, to reinforce hashing security, the SHA-256 algorithm could be refactored with a slow hash algorithm like **aragon2**.

Chapter 7

Results and Analysis

This section outlines the outcomes of the implementations discussed earlier. It begins with presenting some screenshots of the solution accompanied by descriptive commentary. Following this, we detail and analyze the unit tests and security assessments conducted. The section concludes with a comprehensive analysis of these results, highlighting key findings and implications.

7.1 Solution

The user is exposed to four main pages: the connect wallet page, the register page, the login page, and the home page. These are shown in the appendix G.1.

7.1.1 Connect Wallet Page

As the image in the appendix G.1 makes it possible to see, the user connects the wallet with the system. By doing this, he has to choose which account he wants to use during the dApp usage. If the account address already exists in the system, the user is redirected to the Login page; otherwise, he will go to the Registration page.

7.1.2 Registration Page

As the image in the appendix makes it possible to see (G.1), the user registers by giving a unique username, which has to be in lowercase. Further, once the system verifies that the user name is unique, a mnemonic is returned to the user. This popup can be seen in the appendix G.2. This registration process implies the user accepts the transaction execution and consequently

spends money. In this process, the user account is saved in the blockchain alongside the hashed mnemonic. Once the user saves the mnemonic in a safe place, he is redirected to the Home page.

7.1.3 Login Page

As the image in the appendix makes it possible to see G.1, the user gains access to his account if the correct mnemonic is inserted. Once this is performed, the user will be redirected to the Home page. This login process also translates into one transaction that the user has to accept to pay since the user's login status in the blockchain changes.

7.1.4 Home Page

As the image in the appendix makes it possible to see (G.1), on this main page, users have an overview of all files uploaded and an audit log of every action performed over those files. There are multiple actions the users can perform in between them: upload, download, edit, delete, share, verify, and look for details.

These next subsections describe the appendix referring to each of these actions. It is also worth noting that every action in a file translates into one transaction that the user has to accept.

7.1.4.1 Upload File

The user clicks the up arrow in the options to upload a file. A file upload popup is presented, and the user can drop whatever file he intends to. Nevertheless, the only excepted extensions are: ".jpg", "jpeg", ".png", ".gif", ".docx", ".odt", and ".pdf". The screenshots for this can be found in appendix G.3.1. Furthermore, this file upload is added to the Audit Log as it's possible to see in the appendix G.1.

7.1.4.2 Edit File

To edit a file, the user has to select one of the ones showing up in the main view. The edit icon (pencil) will gain color if the user has permission. A popup is presented, and the user then uploads the intended file. As a result, in the main view of the user's files, the old file is substituted by the new edit, and a new entry in the audit log is presented. This can be seen in appendix G.3.2.

7.1.4.3 File Information

By selecting a file, the user can click on the information icon to see the details. This detail includes the file name, version, owner, IPFS CID, State, and Previous Edits. If the user has

download permission to access a file, he can also download previous versions. This is illustrated in appendix G.3.3.

7.1.4.4 Share File / Update Permissions

By selecting a file, if the user has share permissions over it, the share icon is available for the user to click on it. Afterwards, a popup shows up. Here, the user inserts the username of the person to whom the file is shared and the respective permissions. The user has to exist; if the user already has permissions over the file, then they only need to be updated. At the end, a new entry is presented in the audit log, and the file shows (with the selected permissions) in the files view of the user to whom the file was shared. Notice that the user who just received the file can see all past actions performed over the file. This interaction can be seen in the appendix G.3.4.

7.1.4.5 Delete File

By selecting a file, if the user has delete permissions over it, the "trash" icon gets colored. Consequently, deleting a file stops it from showing up in the files view, and a new entry in the audit log is added. This can be seen in the appendix G.3.5

7.1.4.6 Download File

By selecting a file, if the user has download permissions over the file, the user can click on the respective icon. Consequently, the file is downloaded, and a new entry is presented in the Audit Log. This can be seen in appendix G.3.6. It's also possible to download previous versions of the file. This can be seen in appendix G.3.3, section 'Previous Edits'.

7.1.4.7 Verify File

To verify a file, the user only needs to click on the respective icon on the menu. Afterwards, a popup shows up. The user drops the file he wants to ensure it's not tempered, and if there is a file in the system with the same hash as the dropped file, a popup ensuring the file validity is presented, and a new entry in the Audit Log is added. Otherwise, a popup saying the file is not valid is presented. This can be seen in the appendix G.3.7.

7.2 Unit Tests

A **Test-Driven Approach** was taken, and Unit tests were performed to ensure that each component in the system was working in isolation. These tests helped in the development and

refactoring and ensured code robustness and reliability. Tests were performed both in the back-end and in the frontend.

7.2.1 Front end

For the front end, unit tests were only performed over the classes that have **business logic**: "DeleteFileCommand", "DownloadFileCommand", "DropEdit", "DropFileCommand", "Drop-Upload", "SessionExpirationHandler", "ShareFileCommand", "UpdatePermissionsCommand", "UserApp", and "VerifyFileCommand".

A testing framework called **Jest** was used. It allows not only to develop and execute tests but also to generate coverage reports. The appendix H.1 presents the results of running "**npx jest –coverage**". Furthermore, by running the command, a folder called "Coverage" contains the "index.html" with more details on the methods, lines, and branches being tested. With the results, it's possible to see that every test passes, and the coverage percentages of statements, branches, functions, and lines go beyond 90%.

7.2.2 Back end

For the back end, because smart contracts only have business logic, all of them were tested: "AccessControl.sol", "AuditLogControl.sol", "FileRegister.sol", "Helper.sol", "LoginRegister.sol", and "UserRegister.sol".

For unit test development and coverage reports, **HardHats' Testing Framework** was used. The appendix H.2 presents the results of running "**npx hardhat coverage**". Furthermore, by running the command, a folder called "Coverage" is created, and an HTML file is found inside. This file details the methods, lines, and branches being tested. With these results, it's possible to see that every test passes, and the coverage percentages of statements, branches, functions, and lines go beyond 90%.

7.3 Eth-Gas-Resporter

To track the **gas consumption of the smart contracts**, "eth-gas-reporter" was used. To enable real-time gas conversion to any other currency, the **CoinMarketCap** API was needed, [16]. This API provides real-time cryptocurrency price data. Parameters on the "**hardhat.config.js**" had to be added (appendix H.3).

The output of "eth-gas-reporter" execution via the command "npx hardhat test" is detailed in appendix H.3. The resulting table exclusively displays transactions. This is because they are

the only ones with costs since they change the state of the blockchain. Transactions that only consult the blockchain don't have expenses, so they don't show up in this table.

It is worth noting that the result presents the total cost of all calls performed in unit tests. For this reason, to visualize better the cost of one call, the average price in euros is divided by the number of calls and schematized in the table below.

Table 7.1: Cost in EUR of calling a method

Method	Cost/Call
deactivateFile	0.57 EUR
downloadFileAudit	0.43 EUR
editFille	2.42 EUR
recordFileVerification	1.94 EUR
removeUserFileAssociation	0.52 EUR
shareFile	0.50 EUR
updateUserFilePermissions	0.52 EUR
uploadFile	0.48 EUR
recordLogFromAccessControl	0.70 EUR
deactivateFile	0.92 EUR
setAccessControlAddress	0.83 EUR
logOutUser	0.42 EUR
logInUser	0.37 EUR
registerUser	0.23 EUR
setLoginRegisterAddress	0.38 EUR
userRegustered	0.36 EUR

With this resulting table it's possible to see the majority goes around the cents. Nevertheless, this might impact a big company or a user that frequently uses the system. This **proves** that a middle-man mechanism is needed, as pointed out in section 8.1.

7.4 Security Tests

Security testing was also performed to determine how secure the system was. This was executed both on the front end and back end.

7.4.1 Front end

The security coverage tool used for the front end was **EsLint**. When running "npx eslint src/components," a list of errors and warnings appeared.

Appendix I shows an example and solution to a problem found by EsLint. Several rules were added to the EsLint configuration, in between them: 'ESLint-plugin-no-unsantized', 'ESLint-plugin-xss', 'ESLint-plugin-security-node', 'ESLint-plugin-scanjs-rules', 'ESLint-plugin-react-hooks', 'ESLint-plugin-prototype-pollution-security-rules', 'ESLint-plugin-secure-compare', and 'ESLint-plugin-no-wildcard-postmessage'. Furthermore, because some problems being detected were not an issue, they were disabled for the specific situation, see appendix E.9. Once all problems were solved, when executing the command, no output was given, proving that, according to EsLint, no security flaws were found.

7.4.2 Back end

The security coverage tool used for the back end was **Slither**. When executing "**npm run slither**," it outputs a list of problems in red, warnings in yellow, and information in green. More information on this tool can be found in section 3. For the current solution, only the problems and warnings were tried to be solved.

On the output, the only problem showing up concerns a **storage collision** vulnerability [104]. This happens when multiple dynamic arguments are concatenated using "abi.encodePacked()". The message sent by the tool and the code associated with it can be seen in section I.2.1. A more secure "concat()" method was created to solve this problem. This code refactor can be seen in section I.2.1.

Furthermore, there were only two warnings, one related to "**strict equality**" and the other related to "**reentrancy attack**", section 1.1.3. The first was kept the same since it didn't represent a true vulnerability for the pinned code. By following the link mentioned in the warning [106], it was possible to conclude that the same would not happen with the solution. The message for this error can be found in section I.2.2. Unlike the "strict equality" warning, the "reentrancy attack" warning had to be solved. This possible vulnerability was found in many methods. For organizational reasons, only one case is going to be explored here. In section I.2.3 it's possible to see the warning message and the code that originated it. In this case, because the "fileRegister" contract was called before performing the change in the "accessControl" contract (caller) it opens the opportunity for a reentrancy attack. The called contract, "fileRegister", might make additional calls back to the "accessControl" contract before the current function completes execution. In order to solve this reentrancy issue (more details on this vulnerability in section 1.1.3), all the

contract's internal state modifications were performed first, and only then external calls were performed. The solution can be found in section I.2.3.

7.5 Analysis/Conclusion

Several screenshots of the dApp are displayed in the appendix and explained in the current section. To prove the dApps integrity, both unit testing and security coverage were applied.

Only a static analysis approach was performed for unit testing, in which tests were made without executing code. Performing unit tests was harder because methods don't follow the single responsibility principle. More details in 6.6.

As it concerns the **front-end coverage** testing results using **Jest**, it's possible to conclude that good coverage testing was achieved, proving the code is working as expected.

On the **back end**, the testing results obtained using **HardHat** also enable us to conclude that a good test coverage was achieved, which proves the code is executing and working as expected.

On the "**Eth-gas-reporter**," the last column of the output says the average price of executing each transaction. It's possible to see, **prices are high**, reaching 44.93 euros in the edit file. Nevertheless, it is to be taken into account that this method was called 93 times. So, to better understand the price of one transaction, a table was created to schematize it. It's possible to see the majority goes around the cents. Nevertheless, this might have an impact on a big company or a user that frequently uses the system. This **proves** that a middle-man mechanism is needed, as pointed in section 8.1.

Moreover, the front end proved to be secure since **EsLint** no longer displays any warnings or problems.

Lastly, using **Slither**, it was possible to see security vulnerabilities and protect the system against them. It detected one major problem and two warnings. The problem was solved by creating a proper concatenation function, and the warning was solved by executing the transaction that changed the contracts' state first. All problems and warnings found by Slither were successfully solved, giving another security layer to the dApp.

Security tools proved to be important in detecting code vulnerabilities.

Since testing tools cover more than 90% of the code, and security tools no longer show problems, it's possible to conclude the dApp's integrity and consistency.

Chapter 8

Conclusions and Future Work

The present document concerns the dissertation "Identity Management Solution: Security and Trust through Blockchain Technology," oriented by António Monteiro and Pedro Berenguer in the context of the Master in Software Engineering.

The thesis first gives a brief introduction to some **key concepts** used throughout the document, then goes into the analysis of already existing papers and market solutions. This analysis made it possible to position the dissertation in the existing work and define its contribution and innovation.

The main thesis **innovation** is developing a proof of concept file management and verifier solution that prioritizes decentralization over security. It aims to analyze the proposed solution and the challenges faced when developing in a public blockchain, such as Ethereum. It also provides an in-depth analysis of how security and decentralization can go hand in hand when developing a system that aims to store private and confidential user documentation. This aims to provide a unique solution that avoids document forgery and counterfeiting, which usually leads companies to implement complex processes.

The thesis **distinguishes** itself from the other scientific papers by delivering a non-business-specific document manager and verifier solution that prioritizes decentralization over security and by providing an in-depth description of the development process, from requirement elicitation to unit testing and security analysis. This dissertation analyses how decentralized a system can be in a public blockchain without compromising security.

Various **meetings** were held to develop the solution. In these meetings, new requirements were elicited, and old ones refined. To ensure expectations were aligned, requirements **prioritization** was performed. Through this prioritization, it was also possible to determine the

needed requirements to achieve the MVP, Minimum Viable Product. The solution has some **limitations**, in between them, it only interacts with the Ethereum blockchain and with wallets associated with MetaMask, it uses a Mnemonic for user authentication, and every user action is translated into a transaction that has a cost. As for **dependencies**, the system depends on the Ethereum blockchain, MetaMask, and IPFS.

After a careful study of **technologies**, section 3, it was possible to define the ones used during development. Even though the Ethereum blockchain provided a public, distributed environment to deploy smart contracts, it also offered the challenge of data confidentiality, high costs per transaction, and enforced every user action to be a transaction. Several solutions to this problem were also provided and described, such as the usage of a Hierarchical Deterministic Wallet, the usage of ICP, Solana, or Polygon, and a middle-man implementation. Furthermore, during development, it was possible to see that because blockchain is such a young technology, tools used in developing solutions based on it are also constantly evolving. Truffle and MythX were sunset, and the IPFS node of Infura was in maintenance. Solidity also posed challenges in programming because of its lack of features that other languages have.

The developed solution implemented all **'Must-have' requirements** and their main **goals**. A **test-driven development** approach was taken to make the development process more efficient and ensure that previously implemented functionality was functioning well once a new functionality was added.

While developing, as described in chapter 6, **design patterns**, and best development practices were taken into consideration, not only to ensure greater security but also to ensure code maintainability and easier scalability. Nevertheless, these are not perfect and still need better refactoring. In the Entity-Relationship Diagram, it was possible to see that users still own an encrypted symmetric key even though they don't have download permissions. It was also possible to see the exponential growth of the "AccessControl" contract and the "BlockchainWrapper" class. Furthermore, because every user action implied one or more transactions in the blockchain, it was necessary not to follow the 'single responsibility principle,' as would be expected, since it's a good development practice. Nevertheless, with the implementation of a "Command," "Facade," and "Template Method," it was possible to achieve loose coupling, follow the Single Responsibility Principle, and reuse code, achieving greater scalability and code organization.

Regarding **security measures**, section 6.6, multiple were considered to avoid attack vectors.

These measures included blockchain security features, a timeout mechanism, data confidentiality and integrity insurance, input validation, secure authentication and authorization, smart contract robustness, keeping up with the latest technology, and always looking toward a good user experience.

By integrating **security and testing tools**, it was possible to guarantee systems' integrity and robustness. In the front end, testing was performed over classes that had business logic, while in the backend, testing was performed over all smart contracts. Coverage testing tools achieved more than 90% of code coverage, and all problems detected by security tools were successfully solved. Through the output of the Eth-Gas-Reporter, it was also possible to analyze the cost of each transaction, concluding that this solution might become costly for users who frequently interact with the dApp. A potential solution for this would be the implementation of a middleman, as referred in section 8.1. Furthermore, security analysis tools outputted warnings and problems that were solved.

With this successful implementation of the proposed solution and the analysis carried out through the document, it is now possible to answer the proposed research questions defined in section 2.3.4.

- **Question 1: Is it possible to develop a secure, 100% decentralized file manager, tracker, share, and verifier system?**

  The developed solution prioritized **decentralization** over **security**, but because the blockchain itself has its decentralization problems, section 1.1.1, a 100% decentralized system is not possible. Nevertheless, at the application level, several decisions were made to avoid centralization. Because Ethereum is a public blockchain, **decentralization was a challenge**, mainly concerning private key storage. These could not be kept in the blockchains since they would be accessible to everyone in the network. A possible solution could be to integrate a PKI, Public Key Infrastructure, but this solution would centralize the system. An approach similar to the one adopted by MetaMask was implemented in section 1.5, ensuring greater decentralization. This solution implementation can also be seen in section 6.5.2.

- **Question 2: How can a dApp that aims to store important and personal files be public, secure, and decentralized?**

  The content stored in a public blockchain is public and visible to everyone in the network. This poses a problem in ensuring data confidentiality and integrity. A possible solution could have been to use a private blockchain, but this would centralize the system, going

against the main goal of being decentralized. Multiple encryption mechanisms had to be used to overcome this challenge, from symmetric and asymmetric encryption to using Hierarchical Deterministic Wallets (HD). This way, even though this information was stored in a public chain, it was possible to store it securely through encryption mechanisms, enabling users to store important and personal files. These files are encrypted before being stored in the blockchain and in IPFS, as seen in appendix F and explained in section 6.5.4. Not only were files encrypted, but the mnemonic was also hashed, and the private key was always kept safe by being kept on the client side, section 6.5.2.

- **Question 3: What are the systems' limitations, and how do they affect the end user?**

  The system has a few limitations, as explained in section 4.5. Limitations like the mandatory usage of MetaMask might hinder users who don't use this wallet or don't know its existence from adhering to the dApp. Nonetheless, this wallet is easy to configure as a browser extension and easily interacts with the application.

  Furthermore, because mnemonics give the user access to his wallet if he loses it, he loses access to the account forever. Furthermore, the user's account gets compromised if anyone accesses the mnemonic. This might be giving too much responsibility to the user.

  Lastly, every user action translates into a transaction that has a cost. This might hinder the user from adhering to the application since this can be costly if many actions are performed, as proved in section 7.3. The requirement for the user to approve each transaction before it proceeds might deter them from wanting to execute and spend money, as they are continuously reminded of the cost involved.

- **Question 4: What are the best security measures when developing dApp?**

  Security and decentralization go hand in hand, and even though decentralization is the main priority, a final user won't be using a system that's not secure. So, a good balance had to be achieved.

  A good example of a decision where the balance had to be considered was ensuring data confidentiality and integrity. A private blockchain could have been used to achieve this, but this would have hindered decentralization, so Ethereum was the chosen blockchain. However, because Ethereum is a public chain, all data stored in it became public and accessible to everyone in the network, hindering confidentiality and security. A workaround was to implement **symmetric and asymmetric encryption** alongside a **hashing** mechanism and a **Hierarchical Deterministic Wallets (HDW)**. Furthermore, all encryption and decryption were always performed on the client side, and critical information, such as the

users' private key, was always stored in the users' local storage.

Multiple security measures were studied in section 1.1.3 and 1.2.1, and some were applied and explained in section 6.6. In between these security measures, there is data confidentiality and integrity ensured through data encryption, secure authentication performed using a mnemonic, secure authorization through subsequent validations before the user can execute a function, the implementation of timeout mechanisms, and the implementation of robust smart contracts that are not vulnerable to attacks like the reentrancy attack or short address attack. Furthermore, integrating security analysis tools also provides greater security since it detects possible vulnerabilities to which the code might be exposed.

- **Question 5: What are the best tools when developing a dApp?**

  The current solution used the Ethereum blockchain to develop the current dApp. To this end, HardHat was used as a framework in the backend. This tool enabled unit testing and provided a local blockchain to which smart contracts could be deployed, enabling testing smart contracts. Slither was used for security analysis and was able to detect possible security vulnerabilities in the code. Furthermore, "eth-gas-reporter" was used to understand the costs of each transaction and, so, have an estimate of how much the user would pay per action. Solidity was the language used. Solidity proved to be challenging in the sense that it enquired into a more development effort since a lot of built-in functions didn't exist. React, and JavaScript was used for the front end, alongside Jest for unit testing. EsLint was used to detect possible vulnerabilities in code.

  Both the front and back end security and testing tools proved to be useful by enhancing dApps security and its integrity and robustness.

  IPFS was used to store documents, providing security and efficiency.

  While developing, it was possible to experience that technologies are constantly evolving. This should be taken into consideration for future developments.

Although not all requirements were implemented, and there are still code better-ups, the **thesis was successful**. The only not achieved goal, according to the 'Goal Tree', D.3, was using the solution as an identity provider to other apps. The validation strategy was achieved by performing a technology analysis, developing a proof of concept solution, and integrating testing and security tools. The developed solution proved its integrity and robustness by integrating testing and security tools, and all research questions were answered.

8.1 Future Work

The current solution is developed for an individual user to store their documents. Nevertheless, this solution can **grow** and be used inside companies.

A future step is to solve the **transaction cost** issue that may pose a problem for single users and companies to adhere to the system. For this, the chosen Blockchain can be changed, as explained in chapter 3, or if maintaining the Ethereum blockchain, develop a middle-man component that accumulates users' transactions, and once achieving a given threshold, they are sent to the blockchain. A **future study** could define how much transactions' cost would reduce their value when implementing this solution with a middleman.

Furthermore, **not all features were implemented**. Features such as the audit log of the users' actions and the usage of the system as an identity provider could be first implemented. It would also be interesting to explore alternative methods of implementing the Audit Log, such as inferring the necessary data from the blocks' timestamps or other existing on-chain data rather than directly storing the information on the blockchain.

The **design patterns and Entity-Relationship Diagram can also be improved**, and since the **timeout mechanism** implemented in the front end and back end are different, they can both be refactored to be implemented the same way and guarantee greater consistency.

Security can also be improved since the timestamp dependency is not covered, as seen in section 6.7, and the hashing mechanism can also be switched from SHA-256, a fast hashing function, to Aragon2, a slow hashing function. This change would make brute force attacks more difficult since the attacker would inquire into more computational costs and make brute force attacks much more difficult. Further from this, the front end implements a timeout by detecting the user's mouth movement. This can be intrusive for the user, so assuming a solution similar to the one used on the back end would be more interesting. It could also be interesting to understand how mechanisms like **ZK-SNARKS** could be integrating in the current solution to provide further security.

Lastly, to abstract the user from the payment of each action, a mechanism enabling him not to accept each transaction could also be a good approach to improve the system.

Only unit testing was performed. The next step could be to also integrate **penetration testing**, allowing the discovery of new potential attacks and finding hidden vulnerabilities. While unit testing is helpful to ensure the isolated code is working correctly, penetration testing would help detect vulnerabilities by testing the system as a whole.

Integrating **Auditing Techniques** is also fundamental to ensuring system integrity and robustness and controlling attack vectors. Creating CI/CD pipelines and integrating security and testing tools could be the next steps since they ensure that code changes are always scanned for possible vulnerabilities or malfunctioning. Regular security audits and code reviews also help in code maintenance and vector attack control early in development.

References

[1] O. Radu Adrian. The blockchain, today and tomorrow. In *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 458–462, Los Alamitos, CA, USA, sep 2018. IEEE Computer Society.

[2] S. Aggarwal and N. Kumar. Chapter sixteen - hyperledger working model. In S. Aggarwal, N. Kumar, and P. Raj, editors, *The Blockchain Technology for Secure and Smart Applications across Industry Verticals*, volume 121 of *Advances in Computers*, pages 323–343. Elsevier, 2021.

[3] Fahad Alhabardi, Bogdan Lazar, and Anton Setzer. Verifying correctness of smart contracts with conditionals. pages 1–6, 11 2022.

[4] A.A. Andryukhin. Phishing attacks and preventions in blockchain based projects. pages 15–19, 03 2019.

[5] Jyotsna Anthal, Shakir Choudhary, and Ravikumar Shettiyar. Decentralizing file sharing: The potential of blockchain and ipfs. pages 773–777, 05 2023.

[6] Palak Bagga, Ashok Kumar Das, Vinay Chamola, and Mohsen Guizani. Blockchain-envisioned access control for internet of things applications: a comprehensive survey and future directions. *Telecommunication Systems*, 81, 09 2022.

[7] Rajkumar Banoth and Maharshi B. Dave. A survey on decentralized application based on blockchain platform. In *2022 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*, pages 1171–1174, 2022.

[8] Alsadig Bashir and Yahia Fadlalla. A review: Strategies for recognizing forgery in identity documents. *Journal of Imaging*, 04 2021.

[9] Nazanin Benisi, M. Aminian, and Bahman Javadi. Blockchain-based decentralized storage networks: A survey. *Journal of Network and Computer Applications*, 162:102656, 04 2020.

[10] Léo Besançon, Catarina Ferreira da Silva, P. Ghodous, and Jean-Patrick Gelas. A blockchain ontology for dapps development. *IEEE Access*, page 29, 05 2022.

[11] Dipika Bhanushali, Akshara Koul, Sainiranjan Sharma, and Bushra Shaikh. Blockchain to prevent fraudulent activities: Buying and selling property using blockchain. pages 705–709, 02 2020.

[12] H. Bhatia, Indranath Chatterjee, Sheetal Zalte-Gaikwad, and D. Mantri. *Review of Anti-Counterfeit Solutions in Blockchain*, pages 81–96. 06 2023.

[13] Kaushal Binjola, Kevin Joshi, Rajas Bondale, Avani Sakhapara, and Dipti Pawade. Fake passport detection using blockchain. pages 243–246, 12 2022.

[14] BlockHouse. Blockhouse. https://tbtl.com/. Accessed: April 26, 2024.

[15] Clemens Brunner, Ulrich Gallersdörfer, Fabian Knirsch, Dominik Engel, and Florian Matthes. Did and vc: Untangling decentralized identifiers and verifiable credentials for the web of trust. 02 2021.

[16] Coin Market Cap. Coin market cap. https://pro.coinmarketcap.com/. Accessed: May 6, 2024.

[17] Rajasekhar Chaganti, Rajendra Boppana, Vinayakumar Ravi, Kashif Munir, Mubarak Almutairi, Furqan Rustam, Ernesto Lee, and Imran Ashraf. A comprehensive review of denial of service attacks in blockchain ecosystem and open challenges. *IEEE Access*, 10, 09 2022.

[18] Sang-Yoon Chang, Younghee Park, Simeon Wuthier, and Chang-Wu Chen. *Uncle-Block Attack: Blockchain Mining Threat Beyond Block Withholding for Rational and Uncooperative Miners*, pages 241–258. 05 2019.

[19] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. A secure and improved self-embedding algorithm to combat digital document forgery. *Signal Processing*, 89:2324–2332, 12 2009.

[20] Yourong Chen, Hao Chen, Yang Zhang, Meng Han, Madhuri Siddula, and Zhipeng Cai. A survey on blockchain systems: Attacks, defenses, and privacy preservation. *High-Confidence Computing*, 2:100048, 12 2021.

[21] Maliha Chowdhury and Asaduzzaman . A blockchain-based decentralized document authentication system for multiple organizations. pages 269–274, 12 2022.

[22] CryptoWerk. Cryptowerk. https://developers.cryptowerk.com/platform/portal/swagger.html. Accessed: May 6, 2024.

[23] Dipankar Dasgupta, John Shrein, and Kishor Datta Gupta. A survey of blockchain from security perspective. *Journal of Banking and Financial Technology*, 3, 01 2019.

[24] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, PP, 02 2019.

[25] DocuTrack. Docutrack. https://wavpu-oiaaa-aaaam-aabuq-cai.ic0.app/. Accessed: May 6, 2024.

[26] IV encryption. Iv in symmetric encryption. https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.symmetricalgorithm.iv?view=net-8.0. Accessed: May 5, 2024.

[27] Entrust. Entrust website. https://www.entrust.com/. Accessed: May 6, 2024.

[28] EsLint. Eslint front end security tool. https://github.com/eslint-community/eslint-plugin-security. Accessed: June 22, 2024.

[29] Filecoin. Filecoin. https://filecoin.io/. Accessed: June 22, 2024.

[30] filecoint. Filecoin. Accessed: April 26, 2024.

[31] Hrithik Gaikwad, Nevil D'Souza, Rajkumar Gupta, and Amiya Kumar Tripathy. A blockchain-based verification system for academic certificates. In *2021 International Conference on System, Computation, Automation and Networking (ICSCAN)*, pages 1–6, 2021.

[32] Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, and Zhong Chen. Easyflow: Keep ethereum away from overflow, 11 2018.

[33] Nalneesh Gaur. Assessing the security of your web applications. *Linux J.*, 2000(72es):3–es, apr 2000.

[34] Justin Gazsi, Sajia Zafreen, Gaby Dagher, and Min Long. Vault: A scalable blockchain-based protocol for secure data access and collaboration. pages 376–381, 12 2021.

[35] Suman Ghimire and Henry Selvaraj. A survey on bitcoin cryptocurrency and its mining. pages 1–6, 12 2018.

[36] Kishor Datta Gupta, Abdur Rahman, Mohammad Huda, M. A. MAHMUD, and Subash Poudyal. A hybrid pow-pos implementation against 51 12 2019.

[37] Harshavardhan Netha Gurram, Hafeez Mohamad, Abhinav Sriram, and Anjaneyulu Endurthi. A strategy to improvise coin-age selection in the proof of stake consensus algorithm. pages 1–4, 09 2023.

[38] Ahlem Hamdi, Lamia Fourati, and Samiha Ayed. Vulnerabilities and attacks assessments in blockchain 1.0, 2.0 and 3.0: tools, analysis and countermeasures. *International Journal of Information Security*, 10 2023.

[39] Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal*, 10(14):12178–12185, 2023.

[40] Huawei Huang, Jianru Lin, Baichuan Zheng, Zibin Zheng, and Jing Bian. When blockchain meets distributed file systems: An overview, challenges, and open issues. *IEEE Access*, 8:50574–50586, 2020.

[41] ICP. Icp official documentation. https://internetcomputer.org/docs/current/developer-docs/security/general-security-best-practices. Accessed: May 6, 2024.

[42] Inscribe. Inscribe webstie. https://www.inscribe.ai/. Accessed: May 6, 2024.

[43] IPFS. Ipfs. https://ipfs.tech/. Accessed: June 22, 2024.

[44] IPFS. Ipfs desktop app. https://docs.ipfs.tech/install/ipfs-desktop/. Accessed: April 23, 2024.

[45] Mohd Iqbal, Ahmed Obaid, Parul Agarwal, Tabish Mufti, and Ahmed Hassan. *Blockchain Technology and Decentralized Applications Using Blockchain*, pages 555–563. 11 2022.

[46] J Janani, C Krithika, S Keerthika, and J Joshika. Blockchain-based file tracking and data management system for education sector. pages 1–6, 05 2023.

[47] SoonHyeong Jeong and Byeongtae Ahn. Implementation of real estate contract system using zero knowledge proof algorithm based blockchain. *The Journal of Supercomputing*, 77, 10 2021.

[48] Jest. Jest front end testing tool. https://jestjs.io/. Accessed: June 22, 2024.

[49] Doaa Kadhim and Bashar Mahdi. Enhancing the security of the blockchain and the file contents. pages 6–11, 12 2022.

[50] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities, 03 2020.

[51] Hongyue Kang, Xiaolin Chang, Runkai Yang, Jelena Mišić, and Vojislav Misic. Understanding selfish mining in imperfect bitcoin and ethereum networks with extended forks. *IEEE Transactions on Network and Service Management*, PP:1–1, 04 2021.

[52] Smita Kapse, Mrudula Umalkar, Atharva Gajbe, Kaushik Vrudhula, Ritik Gour, and Shital Telrandhe. Blockchain based solution for secured transmission of examination paper. pages 1–6, 12 2022.

[53] Jan Karásek, Radim Burget, and Ondřej Morský. Towards an automatic design of non-cryptographic hash function. In *2011 34th International Conference on Telecommunications and Signal Processing (TSP)*, pages 19–23, 2011.

[54] Sandeepa Kaur, Simarjeet Singh, Sanjay Gupta, and Sangeeta Wats. Risk analysis in decentralized finance (defi): a fuzzy-ahp approach. *Risk Management*, 25, 04 2023.

[55] Atharva Vijay Khade, Harsh Rajesh Patel, and Chirag Modi. Mnemonic phrase management and sim based two-factor authentication (2fa) for mobile wallets in blockchain. In *2023 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*, pages 1–6, 2023.

[56] Hokeun Kim and Edward A. Lee. Authentication and authorization for the internet of things. *IT Professional*, 19(5):27–33, 2017.

[57] Markus Knecht and Burkhard Stiller. Scur: Smart contracts with a static upper-bound on resource usage. 03 2021.

[58] Yujin Kwon, Dohyun Kim, Yunmok Son, Eugene Vasserman, and Yongdae Kim. Be selfish and avoid dilemmas: Fork after withholding (faw) attacks on bitcoin. pages 195–209, 10 2017.

[59] Lukas König, Stefan Unger, Peter Kieseberg, and Simon Tjoa. The risks of the blockchain a review on current vulnerabilities and attacks. 10:110–, 01 2020.

[60] Lukas König, Stefan Unger, Peter Kieseberg, and Simon Tjoa. The risks of the blockchain a review on current vulnerabilities and attacks. 10:110–, 01 2020.

[61] Jihye Lee and Yoonjeong Kim. Preventing bitcoin selfish mining using transaction creation time. pages 19–24, 07 2018.

[62] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020.

[63] N Boris Margolin, Brian N Levine, N Boris Margolin, and Brian Neil Levine. Quantifying and discouraging sybil attacks. Technical report, Technical report, U. Mass. Amherst, Computer Science, 2005.

[64] Rupsingh Mathwale and Ramarao Ramisetty. Blockchain based inter-organizational secure file sharing system. pages 1–5, 03 2023.

[65] Medium. Docutrack. https://medium.com/@dfinity/the-dfinity-foundation-announces-the-open-alpha-release-of-docutrack-1dfdf7e Accessed: May 6, 2024.

[66] Metamask. Metamask. https://support.metamask.io/getting-started/user-guide-secret-recovery-phrase-password-and-private-keys/. Accessed: May 6, 2024.

[67] Mirror. Hierarchical deterministic wallets. https://mirror.xyz/0x9f9f98e28456EEEFC4Af1c990a170e2B0D2d6027/r4R9-kc_KlhwsmpcQVG8b7ENQDkJBjYA1Ob5v0M0l1c. Accessed: May 6, 2024.

[68] Joanna Moubarak, Eric Filiol, and Chamoun Maroun. On blockchain security and relevant attacks. pages 1–6, 04 2018.

[69] Dongjun Na and Sejin Park. Lightweight blockchain to solve forgery and privacy issues of vehicle image data. pages 37–40, 09 2021.

[70] Christopher Natoli and Vincent Gramoli. The balance attack or why forkable blockchains are ill-suited for consortium. 06 2017.

[71] Neeraj Nautiyal, Piyush Agarwal, and Sachin Sharma. Rechain: A secured blockchain-based digital medical health record management system. pages 1–6, 02 2023.

[72] Nethon. Nethon website. https://nethone.com/. Accessed: May 6, 2024.

[73] Kervins Nicolas, Yi Wang, and George Giakos. Comprehensive overview of selfish mining and double spending attack countermeasures. pages 1–6, 09 2019.

[74] Nishara Nizamuddin, Khaled Salah, Muhammad Azad, Junaid Arshad, and Muhammad Habib ur Rehman. Decentralized document version control using ethereum blockchain and ipfs. *Computers Electrical Engineering*, 76, 03 2019.

[75] Kazumasa Omote. Does private blockchain make sense? In *2023 IEEE International Conference on Consumer Electronics (ICCE)*, pages 01–03, 2023.

[76] Oyente. Oyente. https://www.alchemy.com/dapps/oyente. Accessed: June 22, 2024.

[77] Ari Pambudi, Suryari Purnama, Tsara Ayuninggati, Nuke Santoso, and Anggun Oktariyani. Legality on digital document using blockchain technology: An exhaustive study. pages 1–6, 11 2021.

[78] Craig Partridge, Jim Hughes, and Jonathan Stone. Performance of checksums and crcs over real data. *SIGCOMM Comput. Commun. Rev.*, 25(4):68–76, oct 1995.

[79] Shaoliang Peng, Wenxuan Bao, Hao Liu, Xia Xiao, Jiandong Shang, Lin Han, Shan Wang, Xiaolan Xie, and Yang Xu. A peer-to-peer file storage and sharing system based on consortium blockchain. *Future Generation Computer Systems*, 141, 11 2022.

[80] Giuseppe Antonio Pierro and Roberto Tonelli. Can solana be the solution to the blockchain scalability problem? In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1219–1226, 2022.

[81] Pinata. How much does an ipfs pinning service cost. https://www.pinata.cloud/blog/how-much-does-an-ipfs-pinning-service-cost. Accessed: May 6, 2024.

[82] Pinata. Is ipfs forever. https://www.pinata.cloud/blog/is-ipfs-forever. Accessed: May 6, 2024.

[83] Pinata. What is an ipfs pinning service. https://www.pinata.cloud/blog/what-is-an-ipfs-pinning-service. Accessed: May 6, 2024.

[84] S. Porkodi and D. Kesavaraja. Smart contract: a survey towards extortionate vulnerability detection and security enhancement. *Wireless Networks*, pages 1–20, 11 2023.

[85] Yiannis Psaras and David Dias. The interplanetary file system and the filecoin network. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 80–80, 2020.

[86] Swathi Punathumkandi, Chirag Modi, and Dhiren Patel. Preventing sybil attack in blockchain using distributed behavior monitoring of miners. pages 1–6, 07 2019.

[87] Raja Raju, Sandeep Gurung, and Prativa Rai. *An Overview of 51% Attack Over Bitcoin Network*, pages 39–55. 01 2022.

[88] Bharti Rankhambe and Harmeet Khanuja. A comparative analysis of blockchain platforms – bitcoin and ethereum. pages 1–7, 09 2019.

[89] Mahbub Rashid. A review on blockchain security issues and challenges. 10 2021.

[90] Saha Reno, Mamun Ahmed, Saima Jui, and Shamma Dilshad. Securing certificate management system using hyperledger based private blockchain. pages 46–51, 02 2022.

[91] Saha Reno, Shovan Bhowmik, and Mamun Ahmed. Utilizing ipfs and private blockchain to secure forensic information. pages 1–6, 07 2021.

[92] ResistantAI. Resistant ai website. https://resistant.ai/. Accessed: May 6, 2024.

[93] Hossein Rezaeighaleh and Cliff C. Zou. Deterministic sub-wallet for cryptocurrencies. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 419–424, 2019.

[94] Avni Rustemi, Vladimir Atanasovski, Aleksandar Risteski, and Pero Latkoski. Challenges of blockchain in higher education institutions for protection against diploma forgery. pages 1–6, 06 2023.

[95] Alireza Sadeghi-Nasab and Vahid Rafe. A comprehensive review of the security flaws of hashing algorithms. *Journal of Computer Virology and Hacking Techniques*, 19:1–16, 10 2022.

[96] Oiza Salau and Steve A. Adeshina. Secure document verification system using blockchain. In *2021 1st International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS)*, pages 1–7, 2021.

[97] Dionysius Sentausa and David Hareva. Decentralize application for storing personal health record using ethereum blockchain and interplanetary file system. pages 1–6, 09 2022.

[98] Neetu Sharma and Rajesh Rohilla. A multilevel authentication-based blockchain powered medicine anti-counterfeiting for reliable iot supply chain management. *The Journal of Supercomputing*, 80:1–44, 09 2023.

[99] Leyi Shi, Yang Li, Tianxu Liu, Jia Liu, Baoying Shan, and Honglong Chen. Dynamic distributed honeypot based on blockchain. *IEEE Access*, PP:1–1, 05 2019.

[100] Mahendra Shrivas. The disruptive blockchain: Types, platforms and applications. *TEXILA INTERNATIONAL JOURNAL OF ACADEMIC RESEARCH*, pages 17–39, 04 2019.

[101] Sia. Sia. Accessed: April 26, 2024.

[102] Sia. Sia. https://sia.tech/. Accessed: June 22, 2024.

[103] Saurabh Singh, A. S. M. Hosen, and Byungun Yoon. Blockchain security attacks, challenges, and solutions for the future distributed iot network. *IEEE Access*, PP:1–1, 01 2021.

[104] Slither. Econdepacked collision. https://github.com/crytic/slither/wiki/Detector-Documentation#abi-encodePacked-collision. Accessed: May 6, 2024.

[105] Slither. Slither official documentation. https://crytic.github.io/slither/slither.html#api-documentation. Accessed: May 6, 2024.

[106] Slither. Strict equalities. https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities. Accessed: May 6, 2024.

[107] Connecting Software. Cb blockchain seal. https://www.connecting-software.com/cb-blockchain-seal/#. Accessed: May 6, 2024.

[108] Solidity. Solidity. https://soliditylang.org/. Accessed: June 22, 2024.

[109] Solidity. Solidity documentation. https://docs.soliditylang.org/en/v0.8.25/. Accessed: May 6, 2024.

[110] Pallavi Sood and Puneet Bhushan. A structured review and theme analysis of financial frauds in the banking industry. *Asian Journal of Business Ethics*, 9:1–17, 12 2020.

[111] storj. storj. Accessed: April 26, 2024.

[112] Storj. Storj. https://www.storj.io/. Accessed: June 22, 2024.

[113] E. Sujatha, G Naveen Kumar, N Vishnu Vikas, and Madithati Yuvateja Reddy. Decentralized portfolio tracking application on a polygon layer 2 blockchain. In *2023 International Conference on Networking and Communications (ICNWC)*, pages 1–5, 2023.

[114] Hung-Min Sun, Shiuan-Tung Chen, Jyh-Haw Yeh, and Chia-Yun Cheng. A shoulder surfing resistant graphical authentication system. *IEEE Transactions on Dependable and Secure Computing*, 15(2):180–193, 2018.

[115] Jinlei Sun, Song Huang, Changyou Zheng, Tingyong Wang, Cheng Zong, and Zhanwei Hui. Mutation testing for integer overflow in ethereum smart contracts. *Tsinghua Science and Technology*, 27:27–40, 02 2022.

[116] swarm. Swarm. Accessed: April 26, 2024.

[117] Swarm. Swarm. https://swarm.com/. Accessed: June 22, 2024.

[118] Mohammad Tabatabaei, Roman Vitenberg, and Narasimha Veeraragavan. Understanding blockchain: Definitions, architecture, design, and system comparison. *Computer Science Review*, 50:100575, 11 2023.

[119] Shu Takayama, Yuto Takei, and Kazuyuki Shudo. An examination protocol for handling programmable answers using a public blockchain. pages 361–368, 12 2021.

[120] Md Nasim Uddin, Abu Hasnat, Shamima Nasrin, Md Alam, and Mohammad Yousuf. Secure file sharing system using blockchain, ipfs and pki technologies. pages 1–5, 12 2021.

[121] Dejan Vujičić, Dijana Jagodic, and Siniša Ranđić. Blockchain technology, bitcoin, and ethereum: A brief overview. pages 1–6, 03 2018.

[122] Dejan Vujičić, Dijana Jagodic, and Siniša Ranđić. Blockchain technology, bitcoin, and ethereum: A brief overview. pages 1–6, 03 2018.

[123] Vyper. Vyper. https://docs.vyperlang.org/en/stable/. Accessed: June 22, 2024.

[124] yu Wang, Gaopeng Gou, Chang Liu, Mingxin Cui, Zhen Li, and Gang Xiong. Survey of security supervision on blockchain from the perspective of technology. *Journal of Information Security and Applications*, 60:102859, 08 2021.

[125] Yujuan Wen, Fengyuan Lu, Yufei Liu, and Xinli Huang. Attacks and countermeasures on blockchains: A survey from layering perspective. *Computer Networks*, 191:107978, 03 2021.

[126] Canghai Wu, Jie Xiong, Huanliang Xiong, Yingding Zhao, and Wenlong Yi. A review on recent progress of smart contract in blockchain. *IEEE Access*, 10:1–1, 01 2022.

[127] Maximilian Wöhrer and Uwe Zdun. Devops for ethereum blockchain smart contracts. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 244–251, 2021.

[128] Mingyue Xie, Jun Liu, Shuyu Chen, and Mingwei Lin. A survey on blockchain consensus mechanism: research overview, current advances and future directions. *International Journal of Intelligent Computing and Cybernetics*, 16, 09 2022.

[129] Rebecca Yang, Ron Wakefield, Sainan Lyu, Sajani Jayasuriya, Fengling Han, Xun Yi, Xuechao Yang, Gayashan Amarasinghe, and Shiping Chen. Public and private blockchain in construction business process and information integration. *Automation in Construction*, 118:103276, 2020.

[130] Haibo Yi. Securing e-voting based on blockchain in p2p network. *EURASIP Journal on Wireless Communications and Networking*, 2019, 05 2019.

[131] Sheetal Zalte-Gaikwad, H. Bhatia, Rashmi Deshmukh, N. Gupta, and Rajanish Kamat. *Overview of Attack Surfaces in Blockchain*, pages 62–80. 06 2023.

[132] Sheetal Zalte-Gaikwad, H. Bhatia, Rashmi Deshmukh, N. Gupta, and Rajanish Kamat. *Overview of Attack Surfaces in Blockchain*, pages 62–80. 06 2023.

[133] Qixin Zhang. An overview and analysis of hybrid encryption: The combination of symmetric encryption and asymmetric encryption. In *2021 2nd International Conference on Computing and Data Science (CDS)*, pages 616–622, 2021.

[134] Rui Zhang, Rui Xue, and Ling Liu. Security and privacy on blockchain. *ACM Computing Surveys*, 52:1–34, 07 2019.

[135] Zishan Zhao. Comparison of hyperledger fabric and ethereum blockchain. In *2022 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, pages 584–587, 2022.

[136] Mirko Zichichi, Stefano Ferretti, and Gabriele D'Angelo. On the efficiency of decentralized file storage for personal information management systems. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2020.

Appendix A

Blockchain Attacks

Table A.1: Attacks Blockchain: Consensus Mechanism

Group	Attack	Description	Article
Consensus Mechanism	**Majority 51%**	An attacker or a group of attackers (mining pools, [103]) control more than half of the network, manipulating the consensus mechanism.	[87]
	Double Spending	By overcoming the consensus mechanism, an attacker can perform two different transactions using the same fund. It can be performed through 51% attacks, race attacks, Finney attacks, and vector 76 attacks.	[60]
	Alternative History	The attacker creates a private chain and sends a transaction to an honest node. This private chain contains the transaction sent. If the private chain is bigger than the main one, when the attacker releases it, the transaction accepted by the honest node can be invalidated, leading him to lose money.	[131] [12]
	Grinding	Frequent in blockchains that use PoS consensus mechanism. The attacker tries to manipulate the randomness of the process to make him the chosen block validator constantly.	[24]

	Coin Age Accumulation	Frequent in PoS consensus mechanism where nodes are chosen to mine a block based on the amount of stake they have and for how long it is staked, what is called 'coin age'. Given enough time, a malicious node would be able to accumulate a lot of stake, which would be enough for him to take over the network - 'rich getting richer'.	[24]
	Liveness Denial	Form of DoS attack in the PoS protocol. A group of nodes decide to stop producing blocks, leading to the network's failure to execute transactions.	[24]
	Censorship	Because validators can add transactions to a block, they have the power to blacklist certain addresses and decide, for personal reasons, which transactions are going to be added and which won't. Because of nodes performing censorship, some transactions might be delayed or invalidated due to time constraints.	[24]
	Long Range	Both Long-Range attacks and Selfish Mining attacks are attacks where the malicious node mines blocks and keeps them secret. They both fork the main chain and add forged blocks containing different transactions. The difference is that selfish mining attacks don't go back to the genesis block since the computational power needed is too high.	[24]
	Bribery	An attacker tries to manipulate the consensus mechanism by bribing other nodes, gaining control over the networks' decision-making process.	[24]

Table A.2: Attacks Blockchain: Peer-to-Peer

Group	**Attack**	**Description**	**Article**

Peer-to-Peer System	**Selfish Mining**	Attackers, instead of validating a block and broadcasting right way, they keep the block to themselves, creating a private chain. Once the malicious miner has enough blocks in his private chain, he broadcasts it to the network. If a fork is created and the chain uses the "Longest Chain" rule, then the attacker chain will be added to the blockchain, and the attacker will win the block and its rewards. This leads the honest miners to lose the reward and computational power.	[132] [51]
	Fork after withholding	An attacker joins the victim pool and executes a FAW attack. The node submits the proof of work to the pool manager only when another miner outside of the victim pool generates a block. If the pool manager accepts the submitted proof of work of the attacker and propagates it, a fork will be generated. Afterward, the Bitcoin nodes have to select one branch. If the attacker's block is selected, the victim pool receives the rewards, and the attacker is also rewarded from the pool. A miner that belongs to two different pools mines a block and only broadcasts when both pools are available to propagate a valid block. This results in a fork where, independently of which pool that's going to succeed, the miner always wins.	[58] [18]
	Sybil	A malicious node creates multiple fake identities controlled by him. If enough identities are created, the attacker might be capable of performing the 51% attack. This attack is not possible in permissioned blockchains, where identities are needed.	[59] [68]

Eclipse	This attack can be seen as an extension of the Sybil attack, where the attacker controls a set of nodes and surrounds honest nodes, preventing them from connecting with other nodes in the network. This way, because the victim is connected to nodes controlled by the attacker, the victim gets isolated from the network, with its inputs and outputs managed by the attacker. At this point, the victim can't differentiate which nodes are malicious and which aren't. This attack can extend to a 51% attack, where participants can be excluded by surrounding them with the attackers' malicious nodes.	[59]
Wallet	Attackers aim to get wallet credentials, and once they get the users' private key, they can interact with the victim's money. Techniques such as dictionary attacks, vulnerable signatures, and phishing attacks can be used to get wallet credentials.	[132]
BGP Hijacking	Is a routing protocol that connects networks on the internet, routing data packages between networks. This attack is about the illegitimate takeover of IP addresses by corrupting the routing protocol. This way, the attacker can eavesdrop on data, control where data goes, and modify traffic in his favor.	[38]
Balance	In this attack, an attacker with an allowed balance or power tries to disrupt communication among subgroups of miners with similar hashing power. Consequently, the attacker gets information on the victims' communications, being able to carry out various forms of attacks, such as double-spending. This attack aims to delay the network communication between multiple subgroups of nodes with similar mining power.	[38] [70]

Targeted DDoS	Consists of flooding the network with numerous requests beyond what the network handles, making the network unresponsive or temporarily inefficient. This attack might have a big impact since users are not able to perform transactions or perform mining.	[17]
Alien	There is no identity recognition between nodes of the same type of blockchain, leading the nodes of the same type of blockchain to invade and pollute each other. The main reason for this attack is the same chain system doesn't identify non-similar nodes in the communication protocol. An attack method that causes the address pools to pollute each other degrades node communication performance and eventually causes node blockage. Nodes from different chains interact and push their address pool to other blockchain nodes. The victim node eventually broadcasts its address pool to the entire network, causing node congestion, abnormal behaviors of the main network, and other phenomena.	[124]
Pool Hopping	The attacker switches between mining pools, always choosing the one with higher revenue for him. This attack can affect the computing power of the mining pool. This attack can lead to a long wait time for a block to be successfully mined. This is because, when the attack occurs, the entire network's computing power increases considerably; this is, the difficulty value of block mining is excessively high, so the network cannot adjust.	[20]
Uncle Block (UBA)	Extension of the BWA. The difference is that the attacker withholds related blocks in the uncle-block attack. As such, in this attack, the attacker submits all reserved blocks while other miners submit blocks, leading the attacker to get more block rewards since it gets block rewards for the main block and the uncle block.	[20]

	Eavesdropping	An attacker illegally listens on the network to obtain sensitive data being transmitted between two sides. This attack can be considered a passive attack when the attacker spies the messages without changing them or an active attack if the attacker modifies the message content.	[99] [6]

Table A.3: Attacks Blockchain - Application Oriented

Group	Attack	Description	Article
Application Oriented	**Cryptojacking**	Attackers use cryptojacking; this is, attackers use someone's computer to mine cryptocurrency without their knowledge. This is done through malicious links sent through e-mail or by injecting malicious JavaScript code into a web page or an ad. This attack only increases CPU usage and battery drain, making it harder for the victims to detect the attack.	[131]
	Timejacking	The attacker manipulates timestamps by altering the time settings on their node or attempting to influence the time reported by other nodes on the network. As such, the attacker aims to manipulate the networks' perception of time; this is, to control the timing of blocks and transactions to be ahead or behind the actual time.	[131]
	Routing	This attack modifies transactions before sending them to other nodes. In this attack, the attacker changes the network time using **Timejacking** attack and divides the network into two areas that don't communicate with each other. This attack is both a **partition attack** since it divides the network in two, and a delay attack, since it interrupts the propagation of the message and transmits it to the network.	[131]

Repply	When a hard fork happens, users who previously owned coins now own the same amount on the newly created chain. If an investor spends a certain amount of coins from chain A, the attacker is able to intercept the transaction and get the used digital signature. Consequently, the hacker can replicate the transaction to the chain forked from A to chain B, leading to the loss of assets in both chains. Nevertheless, the digital signature must work with the same wallets for this attack to work.	[60]
Length Extension	This attack is based on the predictability of hash functions. In this attack, the attacker is capable of adding data to the original message by only knowing its hash value and length. With this attack, the attacker is capable not only of bypassing security checks but also of creating fraudulent transactions.	[20]
Transaction Malleability	Each transaction has a unique hash used by a miner to track and verify that a transaction has been validated and added to a block. So, this hash can be seen as a way to track transactions within a blockchain. Until a transaction is validated and confirmed, an attacker can change the transaction hash without changing the digital signature, performing a Transaction Malleability Attack. As such, an honest node creates a transaction and sends bitcoins to an attacker. Before this original transaction is validated and confirmed, the attacker changes the transaction ID, changing its hash. Afterward, this transaction is sent to the network, and it gets validated first over the original one, leading the honest node to believe that this transaction was unsuccessful, even though funds were still withdrawn from his account. Once both transactions are validated, the attack is complete.	[131]

	Hash Collision	Happens when two different inputs have the same hash value. This might happen because hash functions have an undefined input length and a predefined output length, making it inevitable for different inputs to have the same hash value.	[95]
	Rug Pull	This attack aims to take advantage of investors, and it is performed by DeFi developers. This attack involves attracting funds into DeFi services, apparently drained by developers who disappear. This might involve activities like hacks or scams.	[54]

Table A.4: Attacks Blockchain: Smart Contracts

Group	**Attack**	**Description**	**Article**
Smart Contracts	**Reentrancy**	The attacker repeatedly calls a function before the previous call has been completed. A contract execution is put on hold when a call to an external contract is performed. This second contract starts having control over the next sequence of events. As such, this second contract can call back again on the first one, leading to a loop or drain of funds. A preventive measure is finishing all internal contact work (updating all contracts' variables) before calling the second contract.	[59] [20]
	Overflow	Attack vulnerability that happens when there are not enough validations to ensure that variables don't exceed their maximum and minimum value. There are various types of overflows: buffer overflow, integer overflow, truncation overflow, and signed overflow.	[125] [32] [115]
	Short Address	This attack happens when the address field is not correctly verified. If a user inserts an address shorter than 20 bytes, the Solidity contract will automatically pad with the right zeros to ensure the address is appropriate.	[125] [89]

	Denial of Service (DoS)	This attack exploits vulnerabilities in the smart contract to make them unavailable. These attacks can be carried out by performing a reentrancy attack or by exploiting the gas limit, entering an infinite loop, and consuming all available gas. A mitigation example can be to optimize the contract code to ensure it does not require excessive gas.	[39]
	Timestamp Dependence	Each block in the blockchain has a timestamp. Miners are capable of manipulating the value of the timestamp. Therefore, "block.timestamp" usage should be avoided; instead, block numbers should be used as they are more predictable and less prone to manipulation.	[39]
	Integer Overflows and Underflows	This vulnerability results in surpassing the fixed range of values. The integer type uint8 has an interval of values between o and 255. If a variable instantiates with this type surpasses 255, the value rests at 0. And if the variable goes lower than 0, the value rests at 255. This can lead to unexpected modifications in the state variables.	[39]

Appendix B

Hierarchical Deterministic Wallets

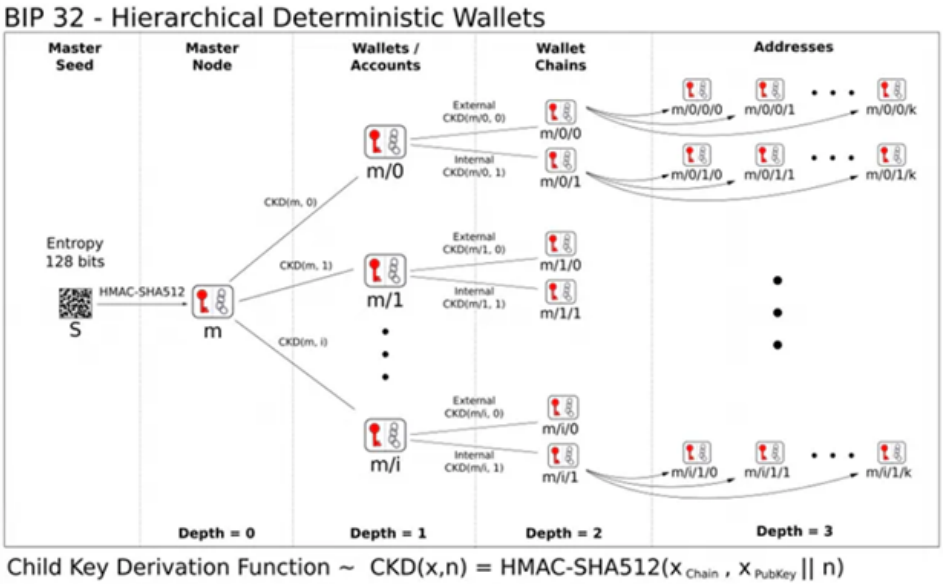


Figure B.1: Hierarchical Deterministic Wallets. [67]

Appendix C

Requirements

This first appendix lists the requirements that were asked for for the current solution.

C.1 Business Requirements

Table C.1: Business Requirements

Requirement	Description	Source	Priority
Rq1	The system should be able to reduce dependence on **intermediaries**.	Sigarra	M
Rq2	The system should be able to **increase** the **efficiency** of the identity verification process.	Sigarra	S
Rq4	The system should be able to use **blockchain** technology.	Sigarra	M
Rq5	The system should be able to **manage personal user files**.	Sigarra	M
Rq5	The system should use mechanisms to **minimize transaction costs** for the user."	Sigarra	C

C.2 Blockchain Requirements

Table C.2: Blockchain Requirements

Requirement	Description	Source	Priority

Rq1	The blockchain should **store user information**.	Sigarra	M
Rq1	The blockchain should **store documents' metadata**.	Sigarra	M
Rq1	The blockchain should **store documents' access control** rules.	Sigarra	M
Rq1	The blockchain should be **public**.	Search	S
Rq1	The blockchain should be **permissionless**.	Search	S

C.3 Non-Funtional Requirements

Table C.3: Non-Functional Requirements

Requirement	Description	Source	Priority
Rq1	The system should be able to ensure the **authenticity** and **integrity** of submitted documents.	Sigarra	M
Rq2	The system should be **transparent** throughout the users.	Sigarra	M
Rq3	The system should be **secure**.	Sigarra	M
Rq4	The system should be able to **protect** the user **data**.	Sigarra	M
Rq5	The system should comply with **data protection regulations**.	Sigarra	M
Rq6	The system should be able to **reduce the risk** of **unauthorized data access/manipulation**.	Sigarra	M
Rq7	The system should **minimize operational costs** while ensuring user satisfaction.	Sigarra	M
Rq7	The system should be **decentralized**.	Sigarra	M
Rq8	The system should provide solid measures to ensure **users' privacy**.	Sigarra	M
Rq9	The system should perform **unit testing**.	Meetings	M
Rq10	The system should perform **test coverage**.	Meetings	M
Rq11	The system should perform **security coverage**.	Meetings	M
Rq12	The system should be able to provide **encryption mechanisms**.	Meetings	M

Rq13	The system should be able to be protected from **web attacks**.	Meetings	M

C.4 Functional Requirements

Table C.4: Functional Requirements

Requirement	Description	Source	Priority
Rq1	The user should have a **unique name**.	Meetings	M
Rq2	The user should have a **username in lowercase**.	Meetings	S
Rq3	The system should be a **web app**.	Meetings	M
Rq4	The user should be able to **register**.	Sigarra	M
Rq5	The user should be able to **log in**.	Sigarra	M
Rq6	The user should be able to **log out**.	Sigarra	M
Rq7	The system should log the user out when the **MetaMask account changes**.	Meetings	M
Rq8	The user should be able to **upload** files.	Sigarra	M
Rq9	When the user has download permissions over a file, the user should be able to **download** the file.	Sigarra	M
Rq10	When the user has download permissions over a file, the user should be able to **download** past file versions.	Meetings	M
Rq11	When the user has delete permissions over a file, the user should be able to **delete** files.	Meetings	M
Rq12	The user should be able to **recover deleted files**.	Market Solutions	C
Rq13	When the user has edit permissions over a file, the user should be able to **edit files**.	Sigarra	M
Rq14	The user should be able to **verify** files.	Sigarra	M
Rq15	The system should consider a file as being **valid** if the user **has** a file with the same content in the 'Active' state.	Meetings	M

Rq16	The system should consider a file as being **invalid** if the user **doesn't have** a file with the same content in the 'Active' state.	Meetings	M
Rq18	The user should be able to see the **details** of a file he can access.	Market solutions	S
Rq19	The **files' details** should include the file name, owner, to whom it was shared with, and previous versions.	Meetings	S
Rq20	When the user has share permissions, the user should be able to **share** files.	Sigarra	M
Rq21	The user should be able to **ask for permissions** over a file.	Meetings	C
Rq22	When the user has share permissions, the user should be able to give **download, edit, delete, and share** file permissions.	Meetings	M
Rq23	The system should be able to generate a **share link** based on the user-defined permissions.	Meetings	C
Rq24	When the user has access to the **shared link** of a file, the user should be able to access it based on the specified permissions under which the link was generated.	Meetings	C
Rq24	When a file is **shared** with the user, the user should be able to see all **past actions** performed over it.	Meetings	S
Rq25	When the user has share permissions, the user should be able to **update permissions** of a user over a file.	Market Solutions	M
Rq26	When the user **removes all permissions** another user has over a file, the association between the other user and the file should be deleted.	Market Solutions	M
Rq27	The user should only be able to **edit** the permissions of **other users** over a file.	Meetings	M
Rq28	The system should consider the **file owner** the user who **uploaded** the file.	Market Solutions	M

Rq29	The file owner should always **have all permissions** over the file.	Meetings	M
Rq30	The system should have an **audit log** that **displays** the user who acted, the file to which the action was applied, the action executed, and the time in which the action was executed.	Market Solutions	M
Rq31	When a user shares or edits a file, the **audit log** should also display the permissions that were given to the user to whom the file was shared with.	Market Solutions	M
Rq32	The system should have an **Audit Log** with a filter file mechanism.	Market Solutions	C
Rq33	The system should have an **Audit Log** with a filter day mechanism.	Market Solutions	C
Rq34	The system should have an **Audit Log** over users' files.	Meetings	M
Rq35	The system should have an **Audit Log** over users' actions.	Meetings	S
Rq36	The system should be able to perform file **versioning**.	Sigarra	M
Rq37	The system should be able to ensure **confidentiality** over files.	Sigarra	M
Rq38	When **accessing other apps**, the user should be able to verify their identity using the dApp.	Sigarra	C

Appendix D

Diagrams and Flowcharts

This appendix stores the flowcharts and diagrams used in the current document.

D.1 Technology Diagram

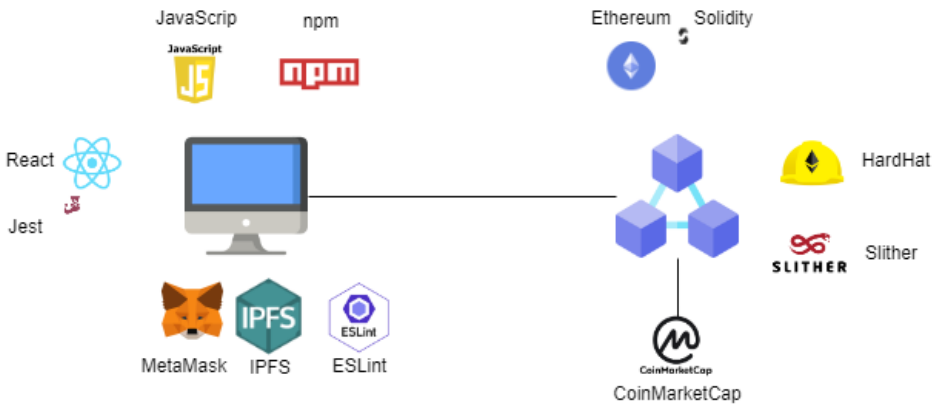


Figure D.1: Technology Diagram

D.2 File Life-cycle: State Diagram

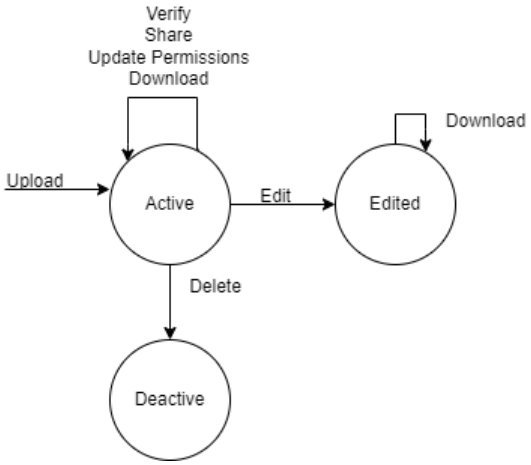


Figure D.2: File Life-cycle

D.3 System Goals: Goal Tree

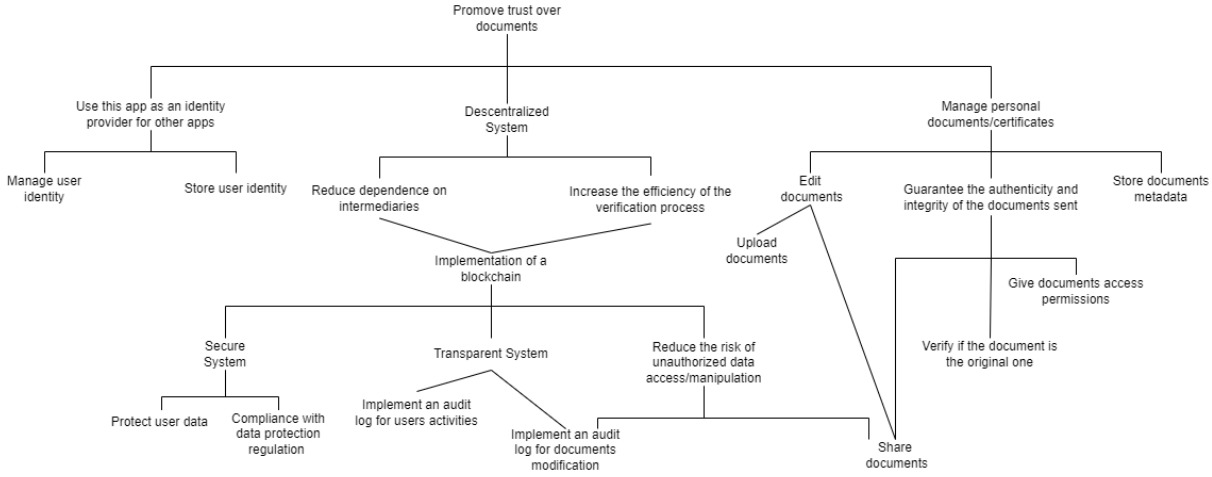


Figure D.3: Goal tree

D.4 System Use Cases: Use-case Diagram

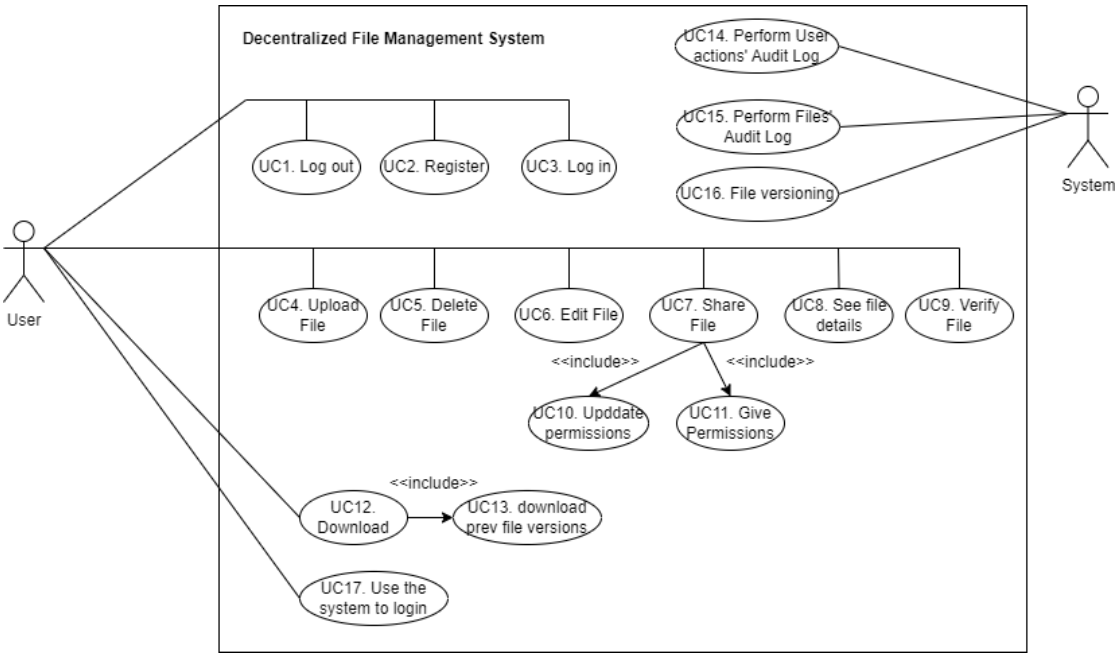


Figure D.4: Use-case diagram

D.5 System Architecture

D.5.1 Abstract Diagram

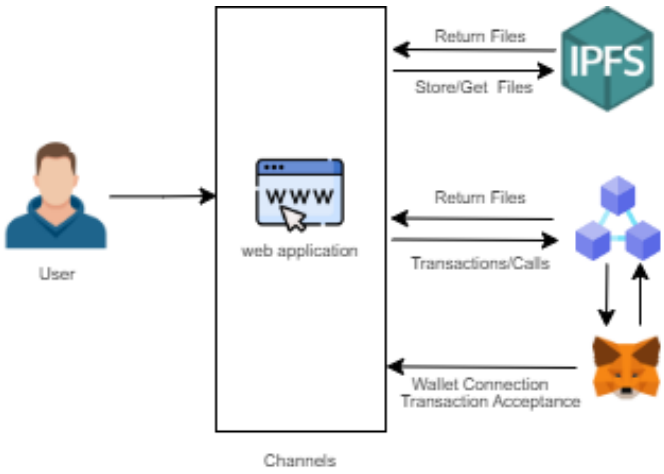


Figure D.5: Abstract Architectural Diagram

D.5.2 Component Diagram

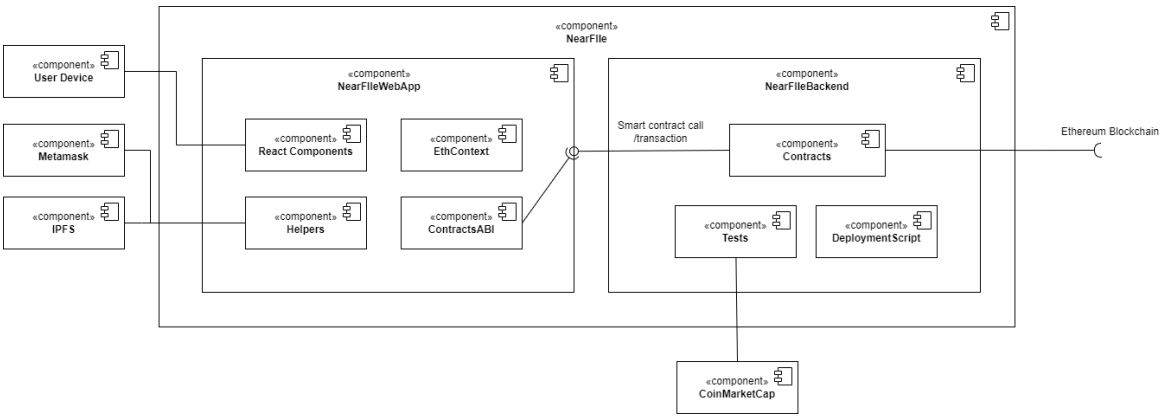


Figure D.6: Component diagram

D.6 Data Management: Entity-Relationship Diagram

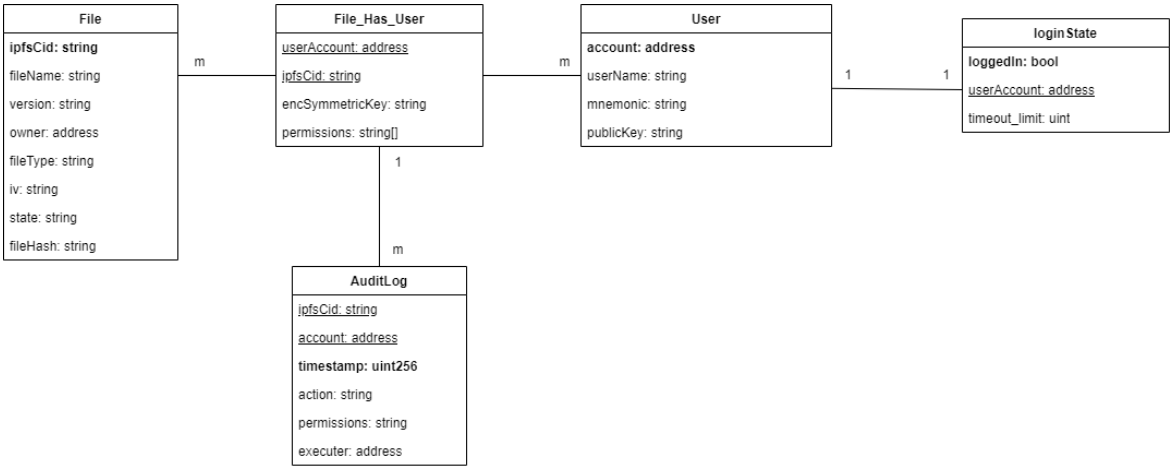


Figure D.7: Entity Relationship diagram

D.7 Interface Interactions: Flowchart

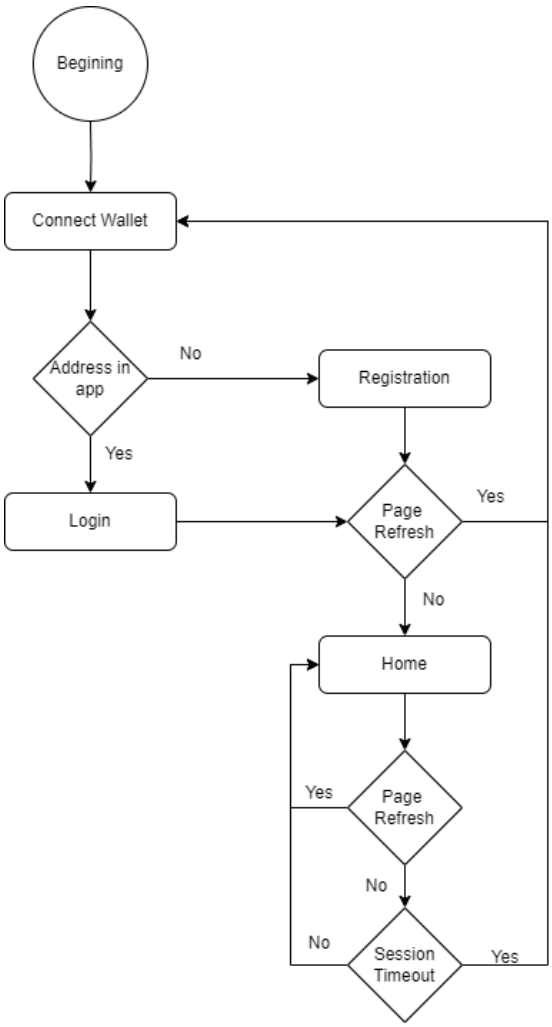


Figure D.8: Interface Interactions

D.8 Front end Design: Class Diagram

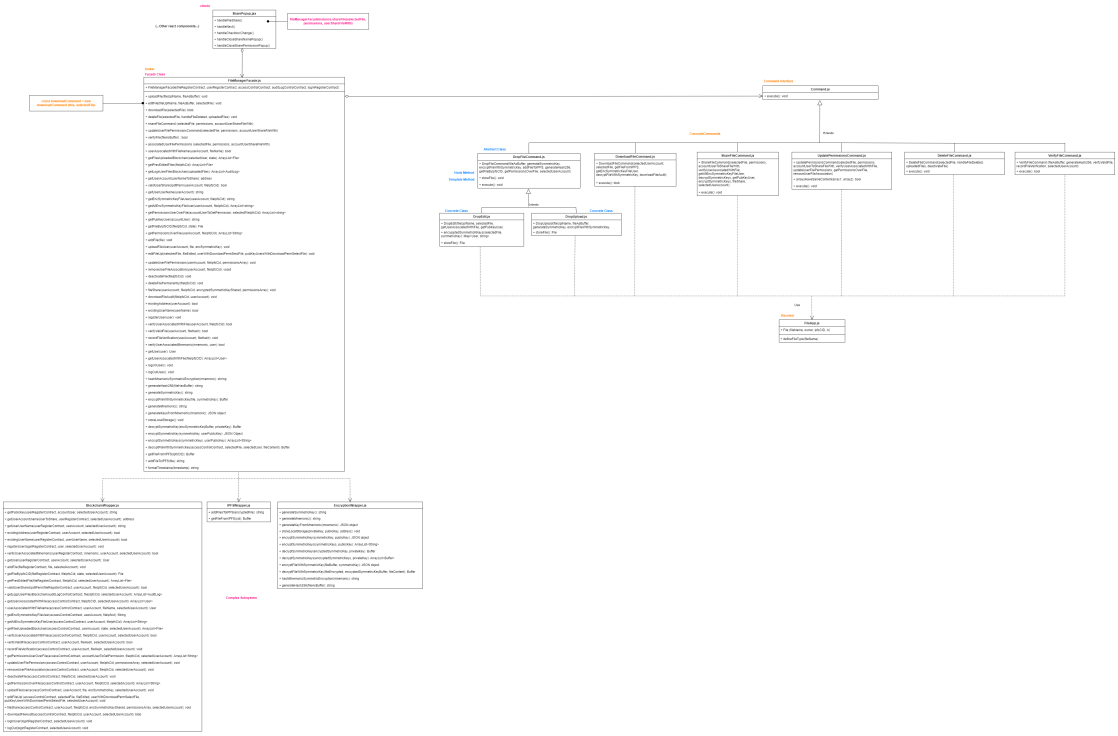


Figure D.9: Front end class diagram

D.9 Back end Design: Class Diagram

Figure D.10: Back end class diagram

D.10 Timeout back end

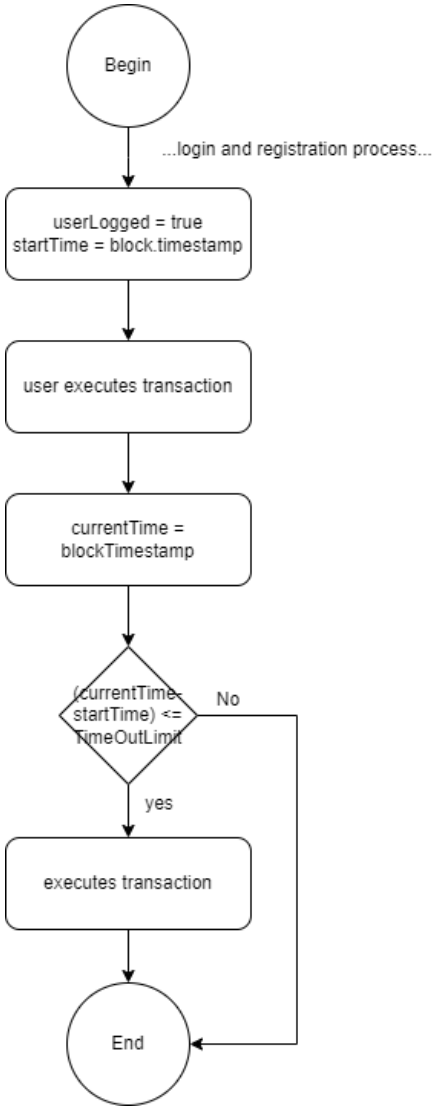


Figure D.11: Timeout Backend

D.11 Timeout front end

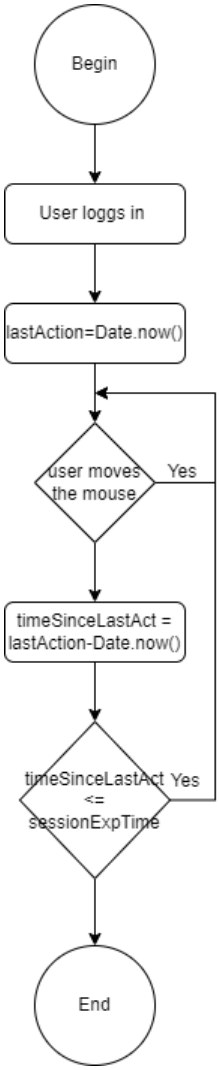


Figure D.12: Timeout Frontend

D.12 Registration and Login: Flowchart

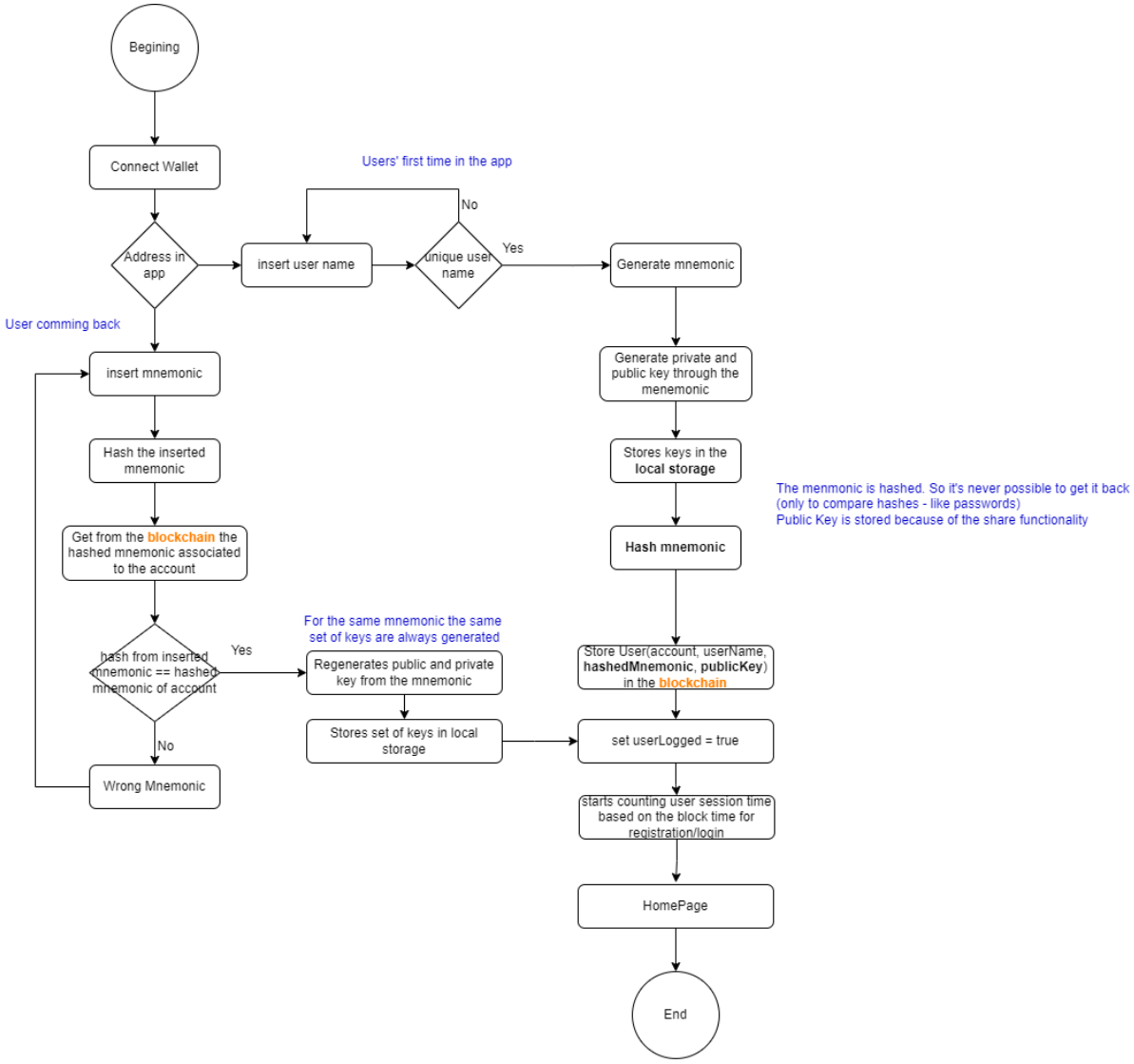


Figure D.13: Registration and Login

D.13 System setup: Flowchart

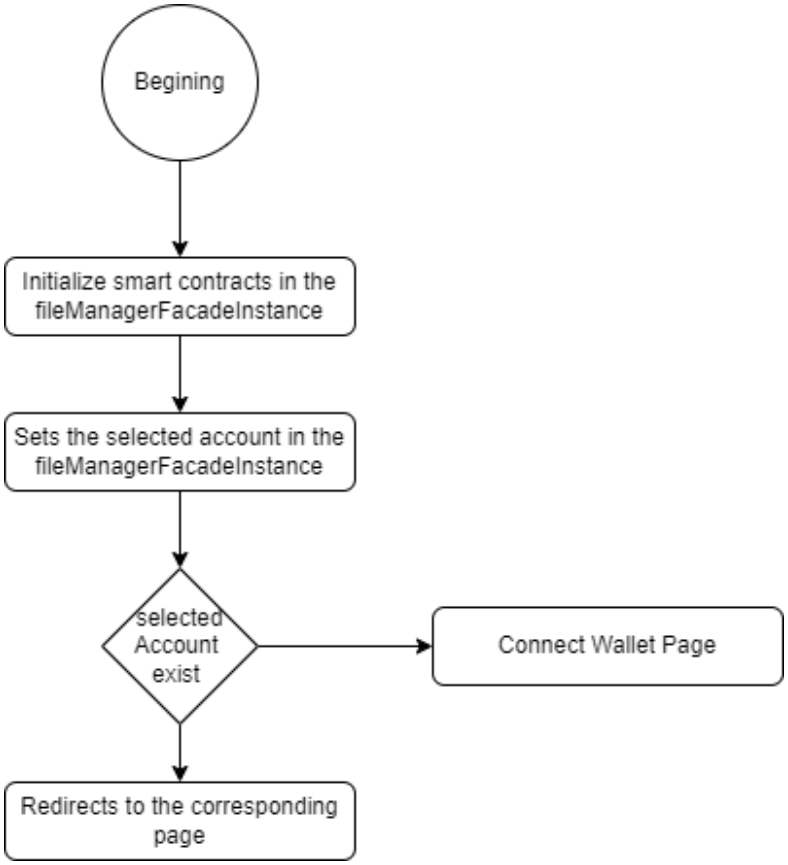


Figure D.14: Set up of the system

D.14 Log out: Flowchart

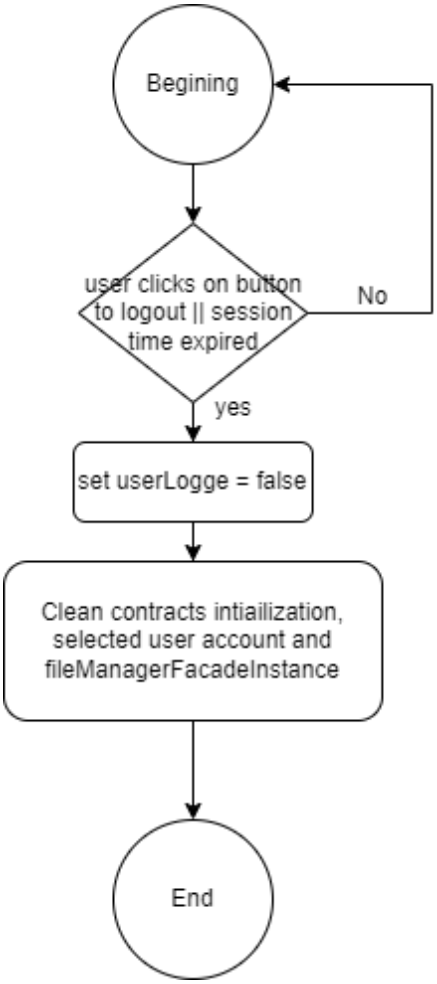


Figure D.15: Log out

D.15 Upload File: Flowchart

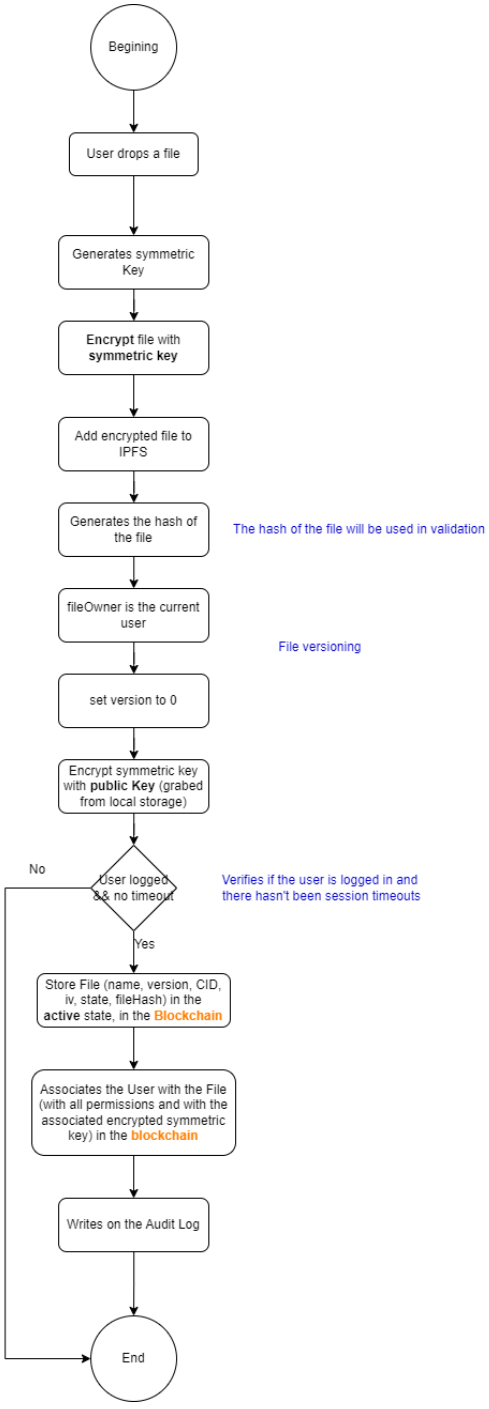


Figure D.16: Upload File

D.16 Edit File: Flowchart

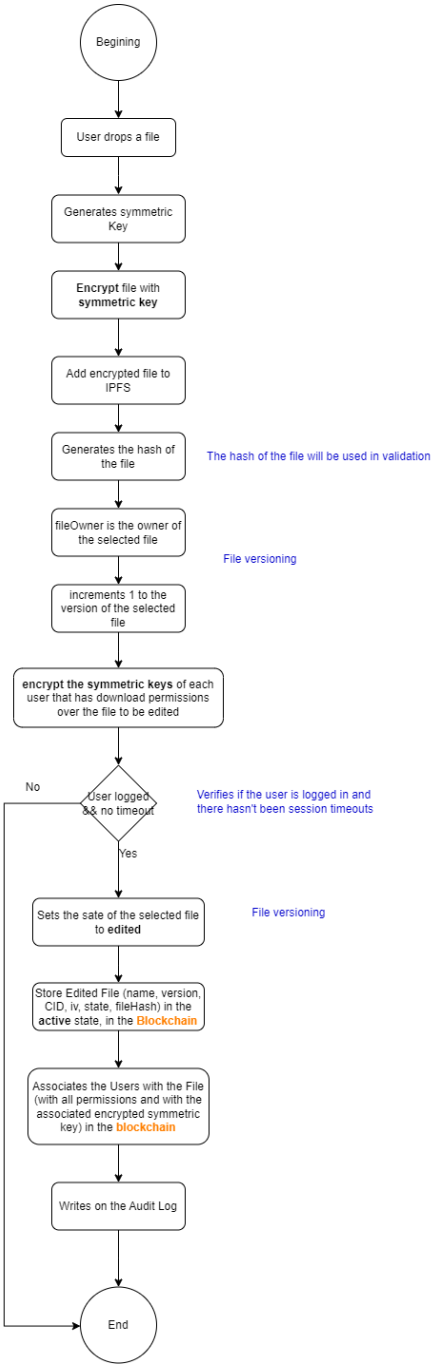


Figure D.17: Edit File

D.17 Download File: Flowchart

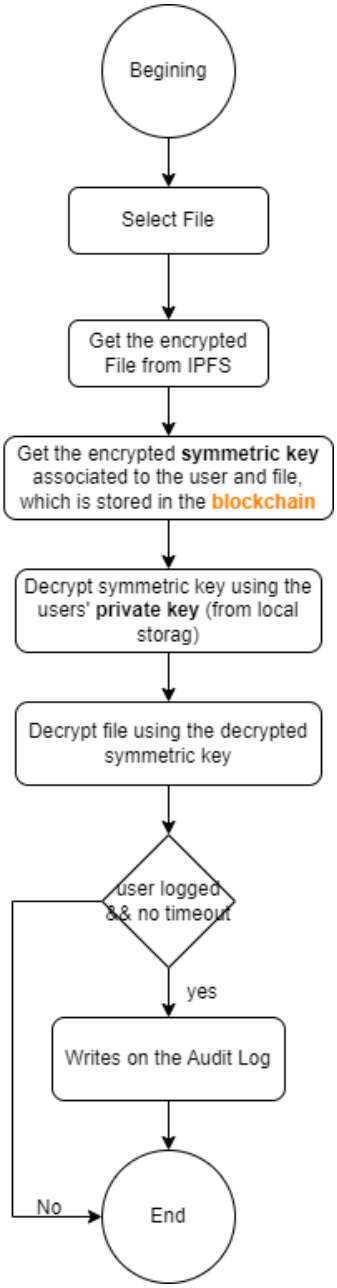


Figure D.18: Download File

D.18 Delete File: Flowchart

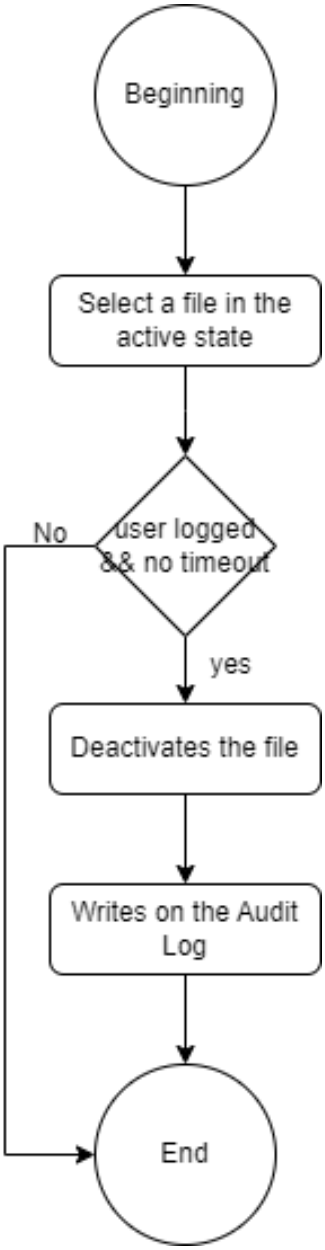


Figure D.19: Delete File

D.19 Share File / Update Permissions: Flowchart

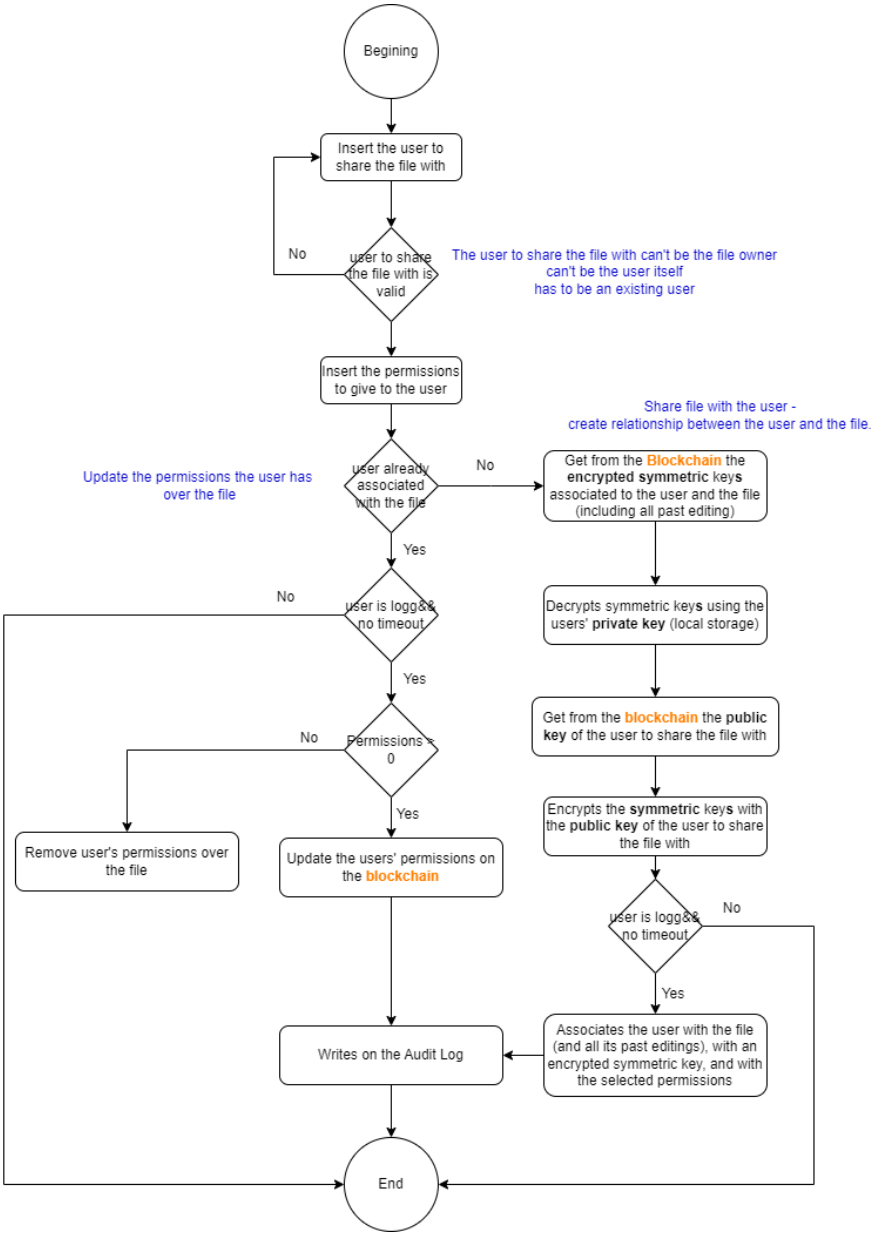


Figure D.20: Share File and Update Users' permissions

D.20 Verify File: Flowchart

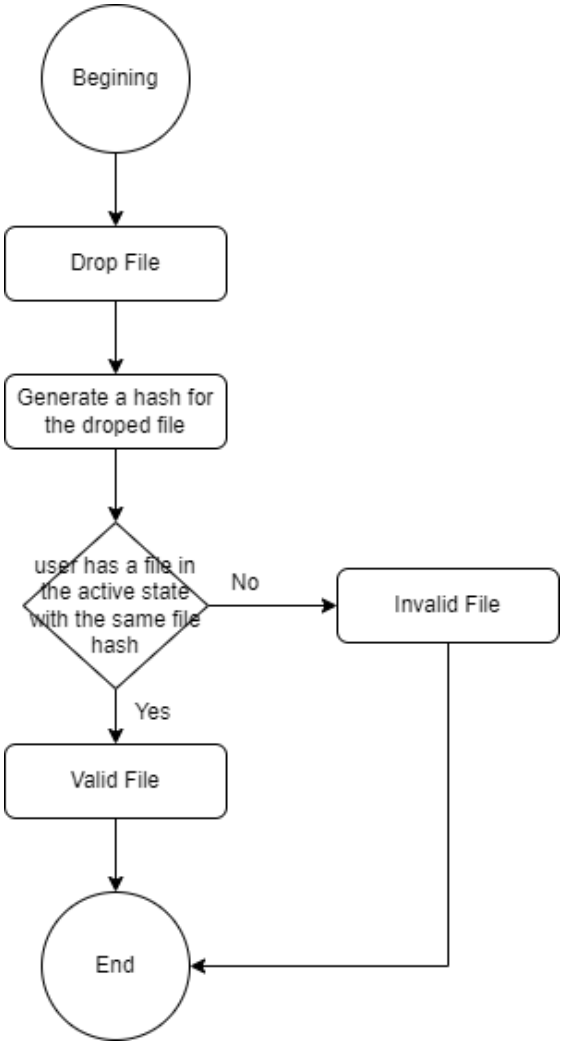


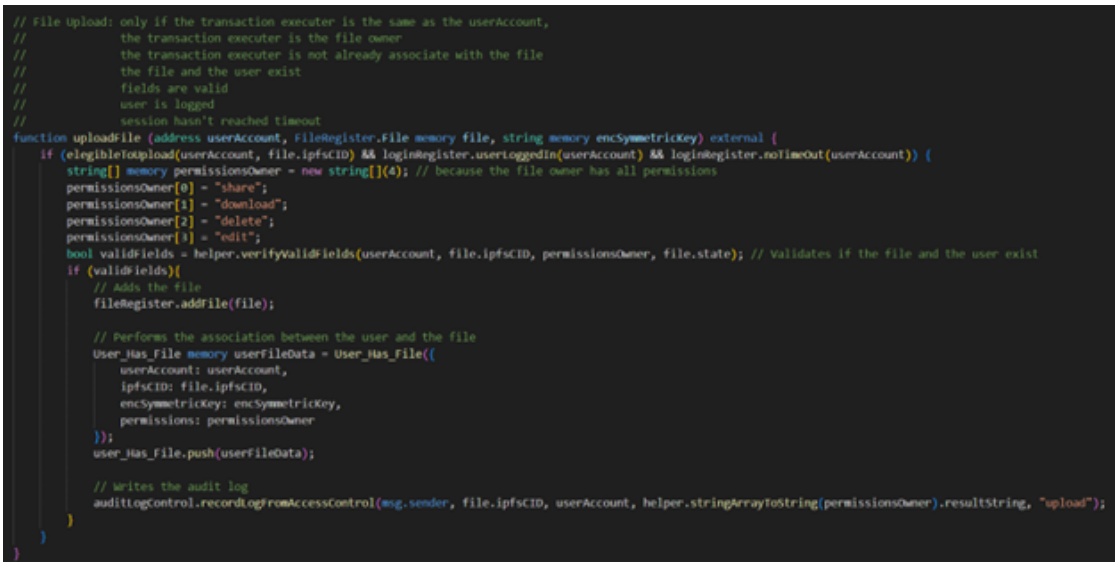
Figure D.21: Verify File

Appendix E

Code Snippets

This appendix stores some code snippets of the developed solution.

E.1 AccessControl.sol



```
// File Upload: only if the transaction executer is the same as the userAccount,
//              the transaction executer is the file owner
//              the transaction executer is not already associate with the file
//              the file and the user exist
//              fields are valid
//              user is logged
//              session hasn't reached timeout
function uploadFile (address userAccount, FileRegister.File memory file, string memory encSymmetricKey) external {
    if (eligibleToUpload(userAccount, file.ipfsCID) && loginRegister.userLoggedIn(userAccount) && loginRegister.noTimeOut(userAccount)) {
        string[] memory permissionsOwner = new string[](4); // because the file owner has all permissions
        permissionsOwner[0] = "share";
        permissionsOwner[1] = "download";
        permissionsOwner[2] = "delete";
        permissionsOwner[3] = "edit";
        bool validFields = helper.verifyValidFields(userAccount, file.ipfsCID, permissionsOwner, file.state); // Validates if the file and the user exist
        if (validFields){
            // Adds the file
            fileRegister.addFile(file);

            // Performs the association between the user and the file
            User_Has_File memory userFileData = User_Has_File({
                userAccount: userAccount,
                ipfsCID: file.ipfsCID,
                encSymmetricKey: encSymmetricKey,
                permissions: permissionsOwner
            });
            user_Has_File.push(userFileData);

            // Writes the audit log
            auditLogControl.recordLogFromAccessControl(msg.sender, file.ipfsCID, userAccount, helper.stringArrayToString(permissionsOwner).resultString, "upload");
        }
    }
}
```

Figure E.1: Upload File

E.2 UserRegister.sol

```
// Verifies if the address exists (if the user already exists)
// Because no infromarion in particular is disclosed and because it only returns true or false,
// This method is also used by the AccessControl.sol
function existingAddress(address account) public view returns (bool) {
    if (users[account].account != address(0)) {
        return true;
    }
    return false;
}
```

Figure E.2: Existing Address

E.3 HardHat.config.js

```
require("@nomicfoundation/hardhat-toolbox");
require('dotenv').config();  6.4k (gzipped: 2.8k)

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.19",
  paths: {
    artifacts: '../client/src/contracts',
  },
  networks: {
    hardhat: {
      allowUnlimitedContractSize: true,
      chainId: 31337,
    }
  },
  gasReporter: {
    currency: 'EUR', // currency which gas costs are reported
    gasPrice: 27,    // gas price to be used for calculating gas costs - adjust according to the fas price in the network
    enabled: true,
    coinmarketcap: process.env.COINMARKETCAP_API_KEY,
    // --- To create a report file uncomment these lines bellow ---
    //outputFile: 'gar-report.txt', // gar report file name
    //noColors: true,
  },
};
```

Figure E.3: HardHat configuration

E.4 Hierarchical Deterministic Wallet Generation

```
import ecies from 'eth-ecies';  134.4k (gzipped: 47.2k)
import { generateMnemonic, mnemonicToSeedSync } from 'bip39';  233.7k (gzipped: 80.6k)
import * as ethUtil from 'ethereumjs-util';  182.5k (gzipped: 54.3k)
import HDKey from 'hdkey';  215.1k (gzipped: 71.5k)
import crypto from 'crypto-browserify';  286.2k (gzipped: 90.2k)

You, 4 weeks ago | 1 author (You)
class EncryptionWrapper {

    // Generate a random symmetric key (for each file)
    static generateSymmetricKey() {
        return crypto.randomBytes(32); // it uses AES-256 algorithm
    }

    // Generates a mnemonic to be asscoiated with a user
    static generateMnemonic() {
        return generateMnemonic()
    }
      You, 2 months ago • updates - a lot (should have done more commits)
    // Generates a set of keys given a mnemonic
    static generateKeysFromMnemonic(mnemonic) {
        const path = "m/44'/60'/0'/0/0"; // derivation path used by metamask
        const seed = mnemonicToSeedSync(mnemonic);
        const hdkey = HDKey.fromMasterSeed(seed);

        const derivedNode = hdkey.derive(path);
        const privateKey = derivedNode.privateKey;
        const publicKey = ethUtil.privateToPublic(privateKey);
        const address = ethUtil.privateToAddress(privateKey);

        console.log("Key Pair generated");
        return {privateKey, publicKey, address};
    }
```

Figure E.4: Code for Hierarchical Deterministic Wallet

E.5 Smart Contract Deployment

```
async function main() {
    async function deployContract(contractName, args = []) {

        const contractFactory = await hre.ethers.getContractFactory(contractName);
        const contract = await contractFactory.deploy(...args);
        await contract.waitForDeployment();
        const contractAddress = contract.target;

        const contractData = {
            address: contractAddress
        }

        // Create a file to store the contract address (hardhat doesn't provide for the frontend)
        const filePath = path.join(__dirname, `../../client/src/contracts/${contractName}_ContractAddress.json`);
        fs.writeFileSync(filePath, JSON.stringify(contractData, null, 2));

        console.log(
            `${contractName} deployed to ${contractAddress}`
        );

        return contractAddress;
    }

    async function addAdressToFile(contractAddress, fileName) {
        const contractData = {address: contractAddress};
        const filePath = path.join(__dirname, `../../client/src/contracts/${fileName}_ContractAddress.json`);
        fs.writeFileSync(filePath, JSON.stringify(contractData, null, 2));
        console.log(`${fileName} deployed to ${contractAddress}`);
    }

    // Deploys contracts
    const helperContractAddress = await deployContract("Helper");
    const accessControlContractAddress = await deployContract("AccessControl", [helperContractAddress]);
```

Figure E.5: Smart Contracts Deployment

```
    // Gets the deployed AccessControl contract
    const accessControlContract = await hre.ethers.getContractAt("AccessControl", accessControlContractAddress);
    // Get the address of the contracts deployed by the AccessControl
    const auditLogControlContractAddress = await accessControlContract.getAuditLogControlAddress();
    const fileRegisterContractAddress = await accessControlContract.getFileRegisterAddress();
    const userRegisterContractAddress = await accessControlContract.getUserRegisterAddress();
    const loginRegisterContractAddress = await accessControlContract.getLoginRegister();
    addAdressToFile(auditLogControlContractAddress, "AuditLogControl");
    addAdressToFile(fileRegisterContractAddress, "FileRegister");
    addAdressToFile(userRegisterContractAddress, "UserRegister");
    addAdressToFile(loginRegisterContractAddress, "LoginRegister");

    const fileRegisterContract = await hre.ethers.getContractAt("FileRegister", fileRegisterContractAddress);
    // Sets the needed variables for each contract
    await fileRegisterContract.setAccessControlAddress(accessControlContractAddress);        // sets the address
}       You, 3 months ago • contract to store user, and added on the contra…
```

Figure E.6: Smart Contracts Deployment (continuation)

E.6 Upload File

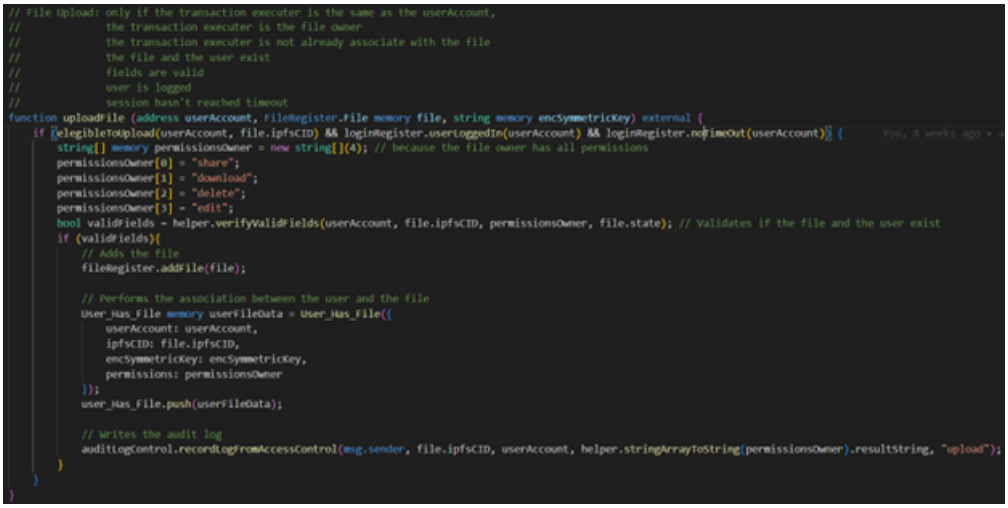


Figure E.7: Upload File - permissions hardcoded.

E.7 Edit File

```
// Edits a file - like an upload
// Edit a file if: the transaction executer is the accessControl address
// In here: the file owner is the same
//          the previousIpfsCid is the one from which the file is being edited from
//          the file version is incremented to one
//          the file state is now edited but the new edited file is in the state active
function editFile(File memory selectedFile, File memory newFile) external {
    if (msg.sender == accessControlAddress &&
        selectedFile.owner == address(uint160(selectedFile.owner)) &&
        selectedFile.owner != address(0)
    ) {
        files[selectedFile.ipfsCID].state = "edited";

        newFile.owner = selectedFile.owner;
        newFile.prevIpfsCID = selectedFile.ipfsCID;
        newFile.version = selectedFile.version + 1;
        newFile.state = "active";

        files[newFile.ipfsCID] = newFile; // adds the new file
        ipfsCids.push(newFile.ipfsCID);
    }
}
```

Figure E.8: Edit File - arguments validation.

E.8 Timeout not Performed

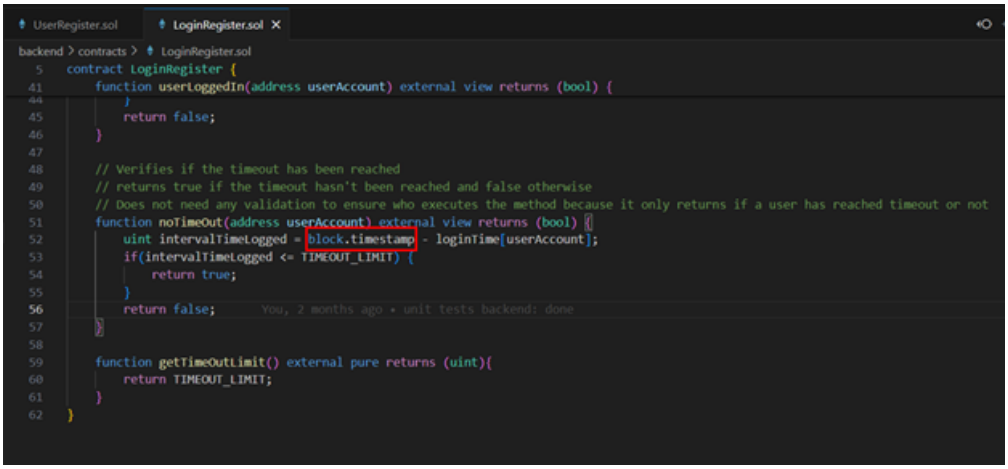


Figure E.9: Timeout mechanism vulnerable to attacks (block.timestamp)

E.9 EsLint Rule

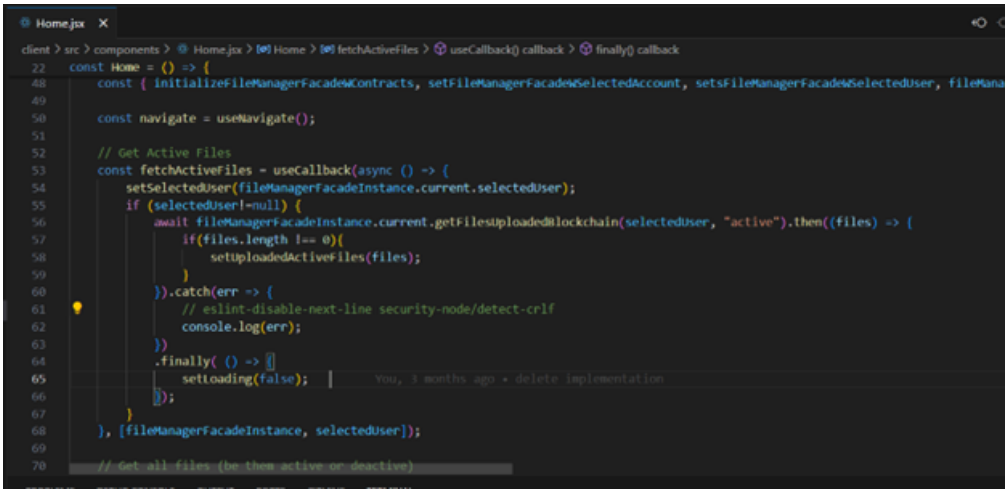


Figure E.10: EsLint rule being ignored for the specific case.

E.10 Share File - Validation

```
class ShareFileCommand extends Command {

    async execute(){
        // Gets only the selected permissions
        // eslint-disable-next-line security/detect-object-injection
        const permissionsArray = Object.keys(this.permissions).filter(key => this.permissions[key]);

        // If the user is already associated with the file
        const userIsAssociatedWithFile = await this.verifyUserAssociatedWithFile(this.accountUserToShareFileWith, this.selectedFile.ip
        if (userIsAssociatedWithFile) {
            console.log("It was called 'ShareFileCommand' but the user is already associated with the selected file.");
            return;
        }

        // Get the encrypted symmetric key that a user has over a file (including all the previous editings of the file)
        var result = await this.getAllEncSymmetricKeyFileUser(this.selectedUserAccount, this.selectedFile.ipfsCID);
        if(!result.success){
            console.log("something went wrong while trying to get the encrypted symmetric keys of the given file and user");
            return;
        }       You, 3 months ago • refactor
        var encSymmetricKeys = result.resultStrings;

        // Decrypts the given symmetric keys, using the Users' public key
        var decSymmetricKeys = await this.decryptSymmetricKeys(encSymmetricKeys, localStorage.getItem('privateKey'));

        // Get the public key of the user to share file with
        result = await this.getPubKeyUser(this.accountUserToShareFileWith);
        if (!result.success) {
            console.log("Something went wrong while trying to get the public key of the user.");
            return;
        }
        var publicKeyUserToShareFileWith = result.resultString;

        // Encrypts the symmetric keys with the users' public key
        var encryptedSymmetricKeysShared = await this.encryptSymmetricKeys(decSymmetricKeys, publicKeyUserToShareFileWith);
```

Figure E.11: Share File - Validates if the user is already associated with the file, before executing the code.

Appendix F

IPFS

This appendix stores screenshots on IPFS storage.

F.1 IPFS storage

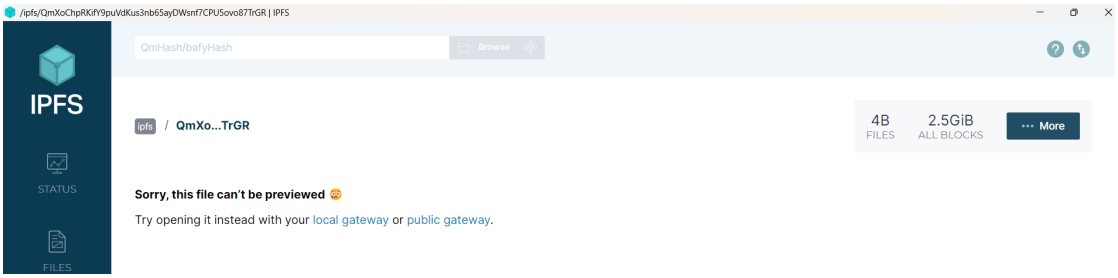


Figure F.1: File encrypted in IPFS

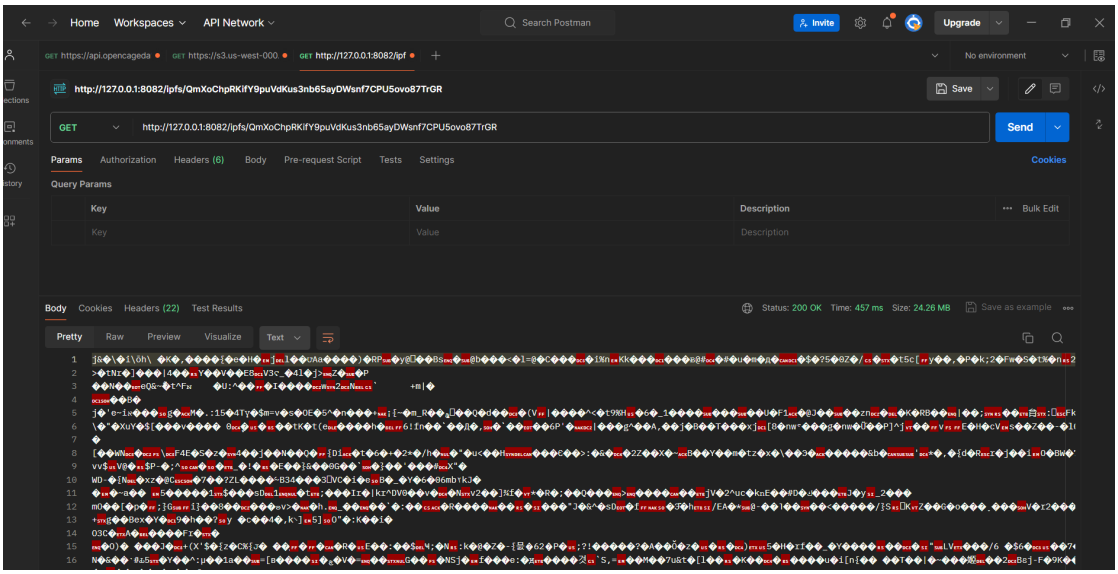


Figure F.2: File encrypted in the IPFS - Postman

Appendix G

Solution

This appendix stores some screenshots of the developed solution.

G.1 Main Pages

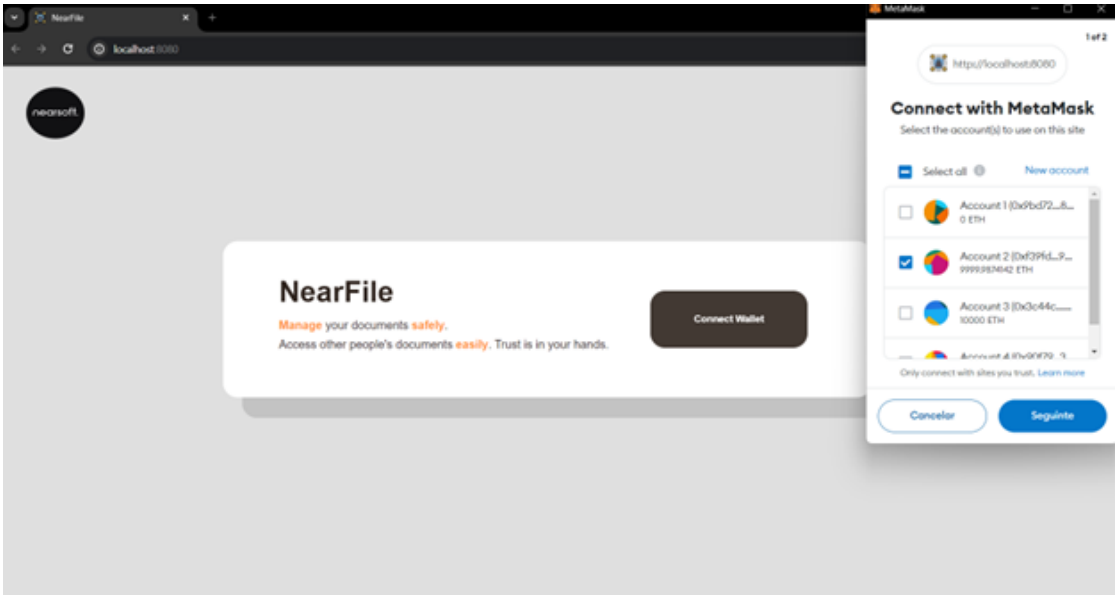


Figure G.1: Connect Wallet Page

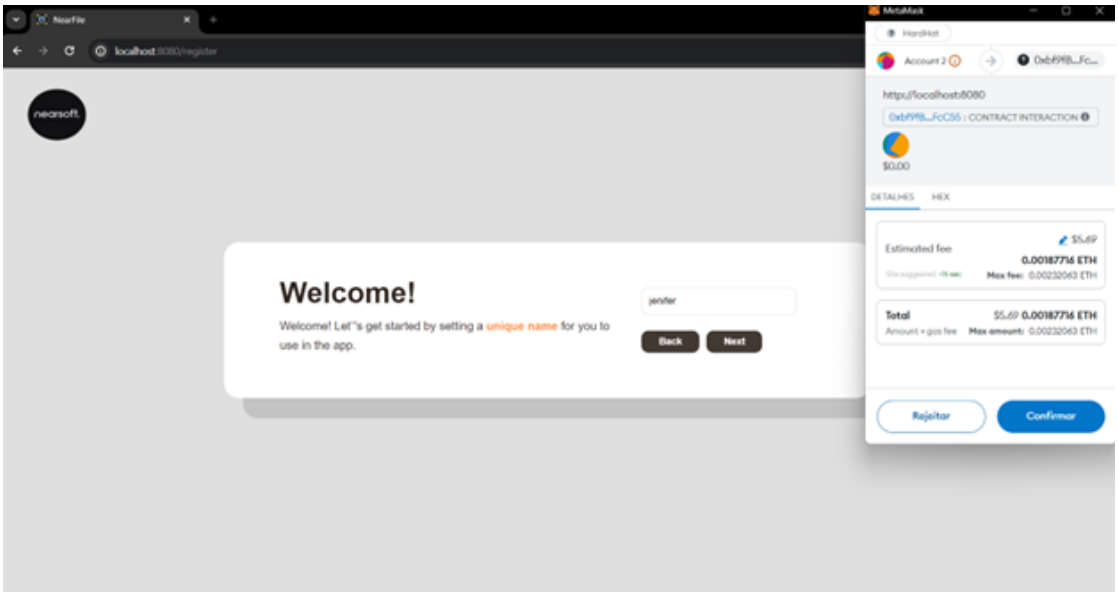


Figure G.2: Register Page

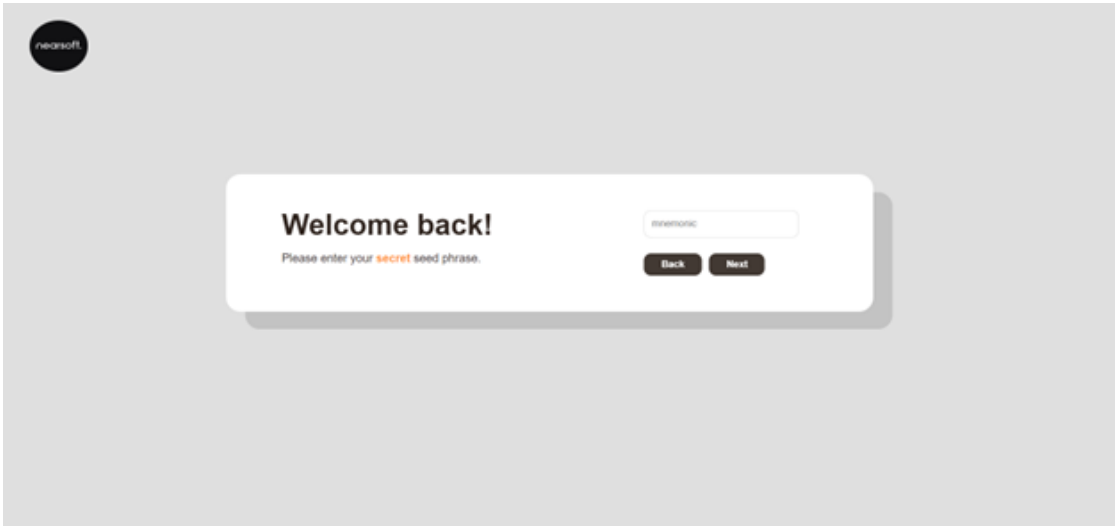


Figure G.3: Login Page

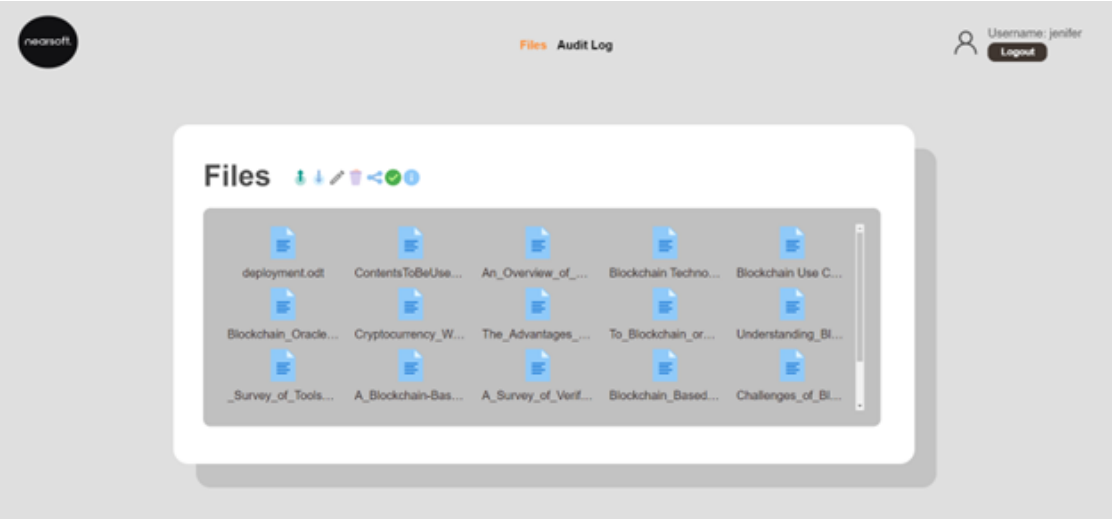


Figure G.4: Home Page

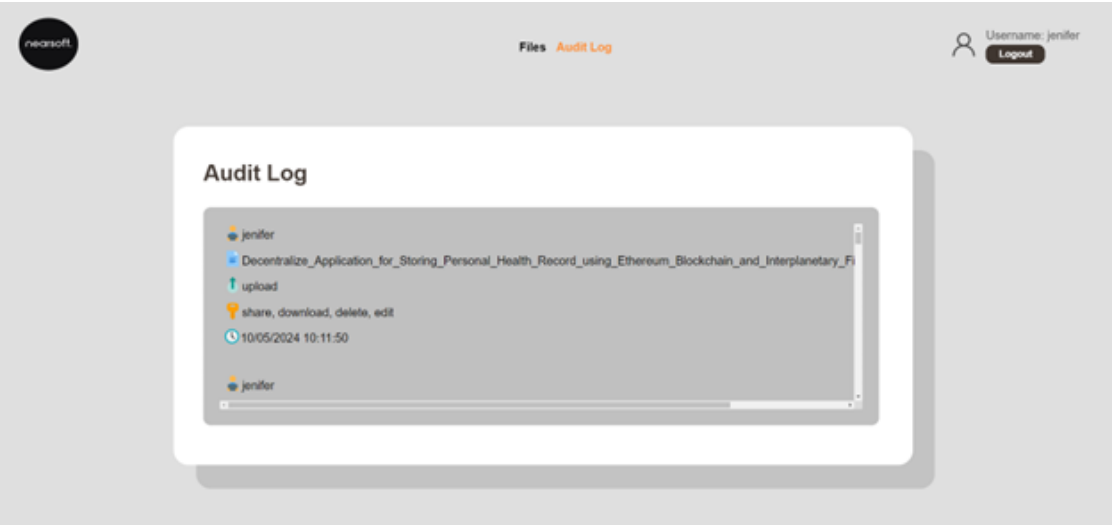


Figure G.5: Home Page (continuation)

G.2 Registration Page

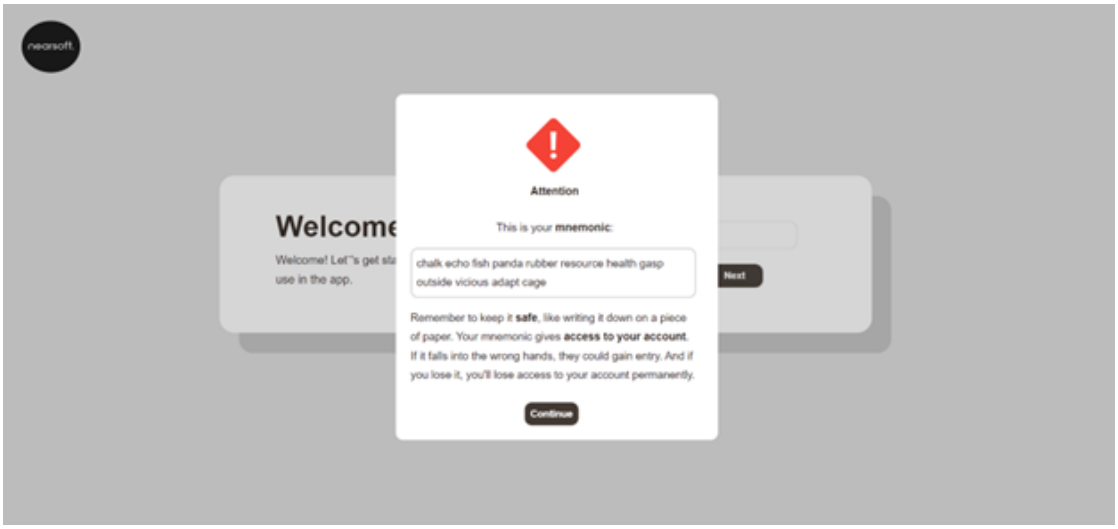


Figure G.6: Mnemonic Popup

G.3 Home Page

G.3.1 Upload File

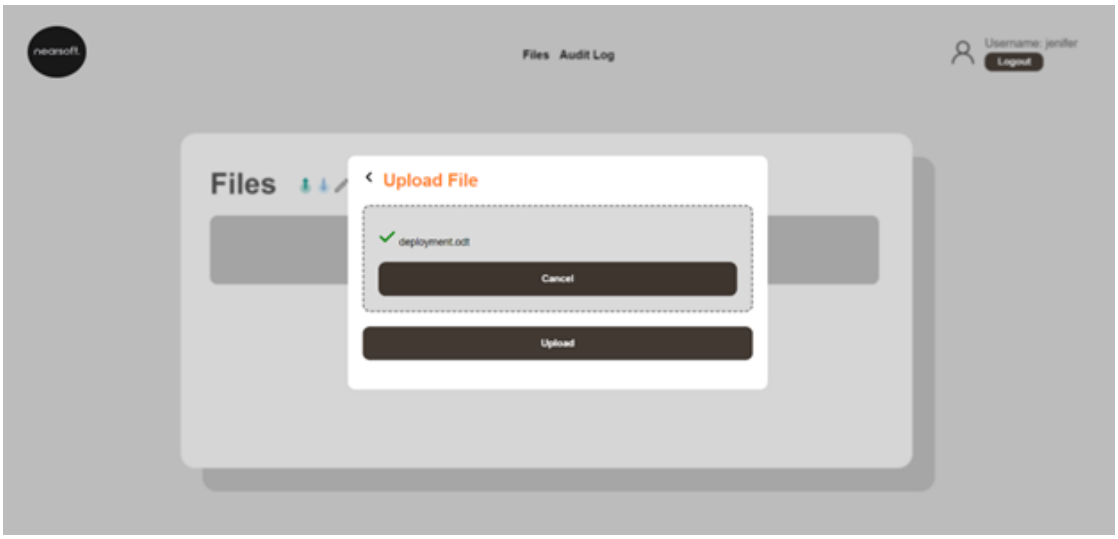


Figure G.7: Upload File

Figure G.8: Upload File - Invalid Extension

G.3.2 Edit File

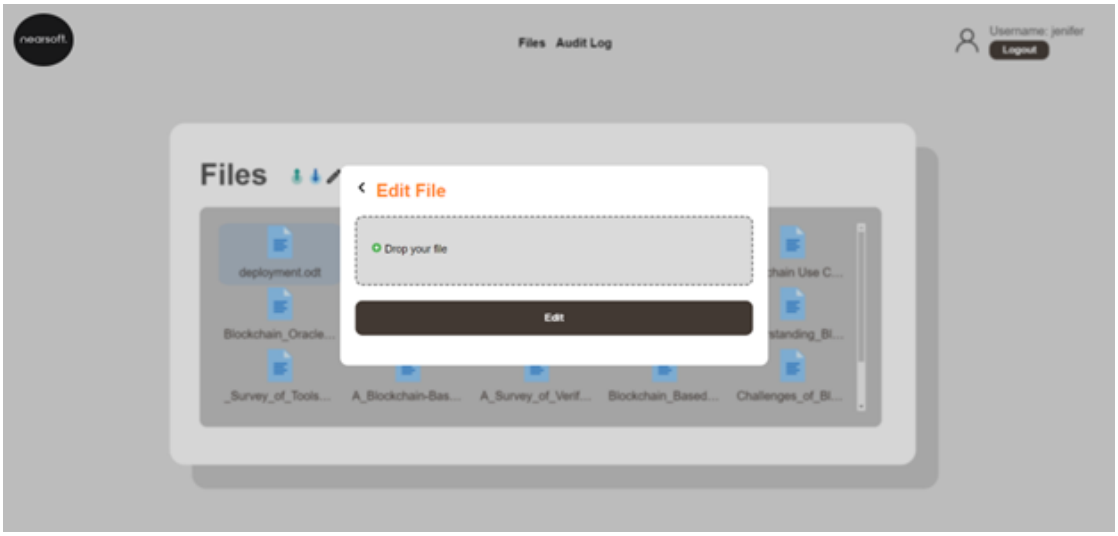


Figure G.9: Edit File

Figure G.10: Edit File - Old File substituted by the new one

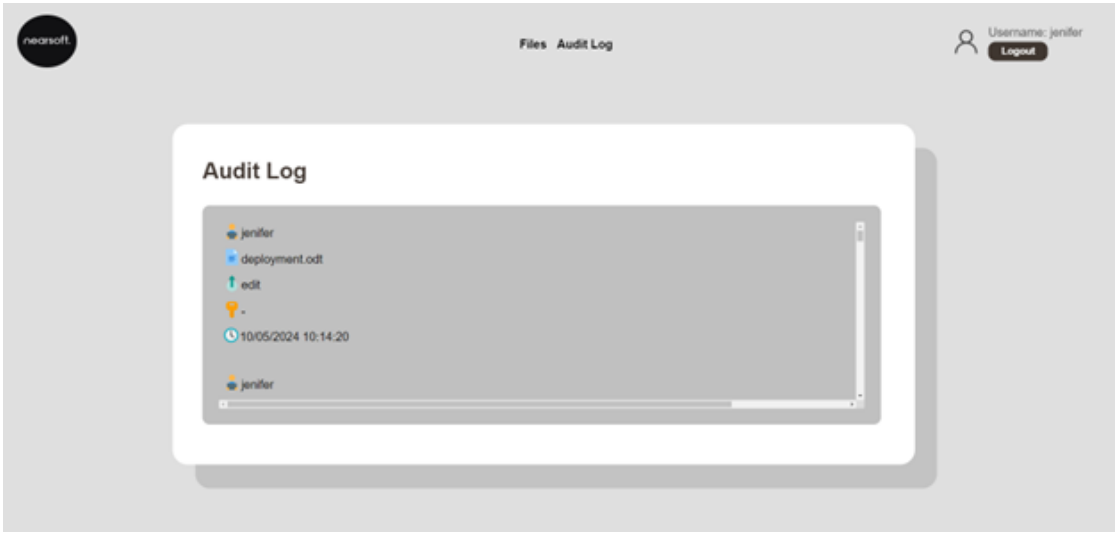


Figure G.11: Edit File - New entry in the audit log

G.3.3 File Information

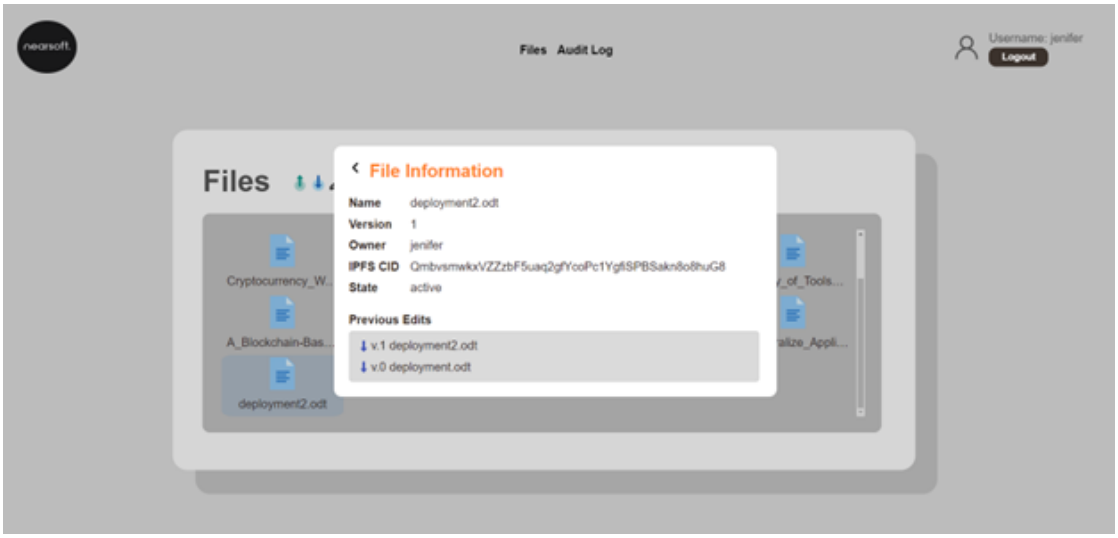


Figure G.12: File Information

G.3.4 Share File

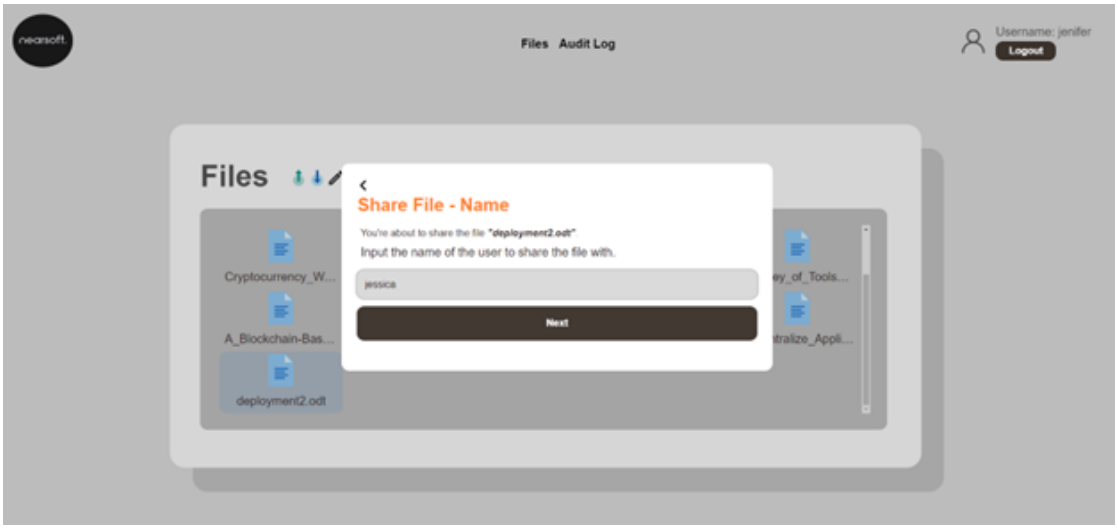


Figure G.13: Share File - name user to share the file with.

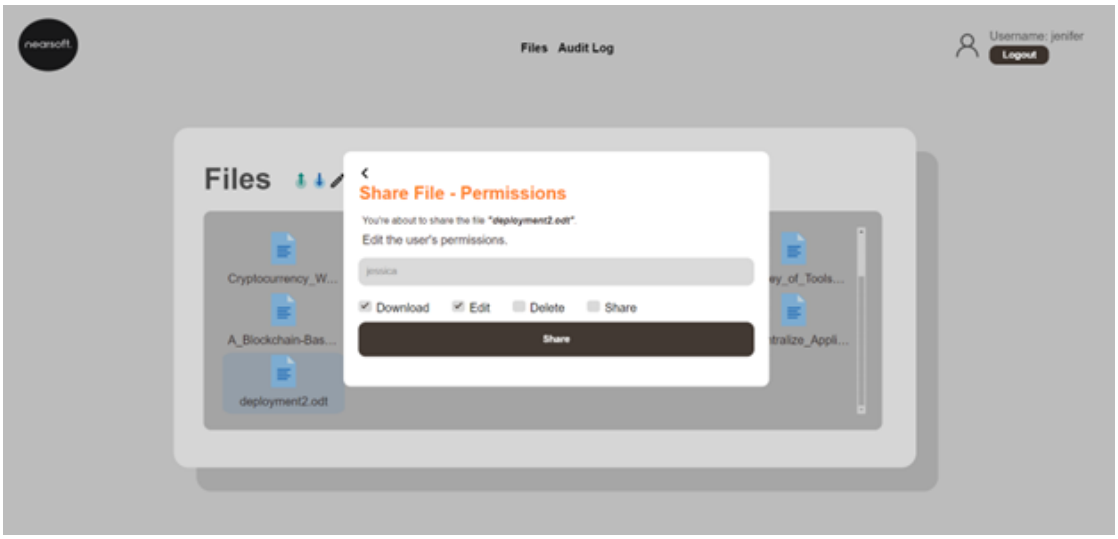


Figure G.14: Share File - permissions

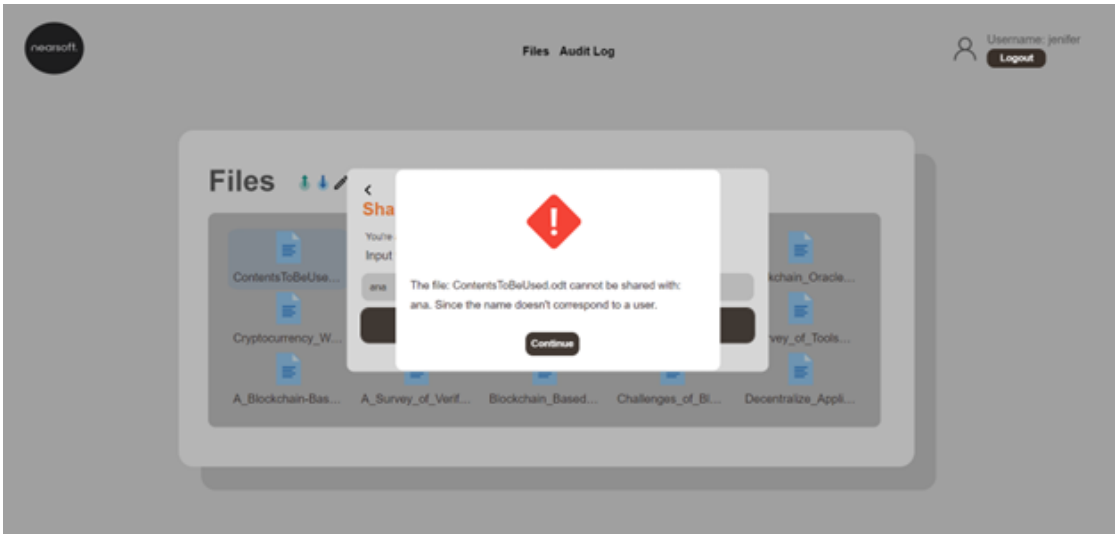


Figure G.15: Share File - user doesn't exist

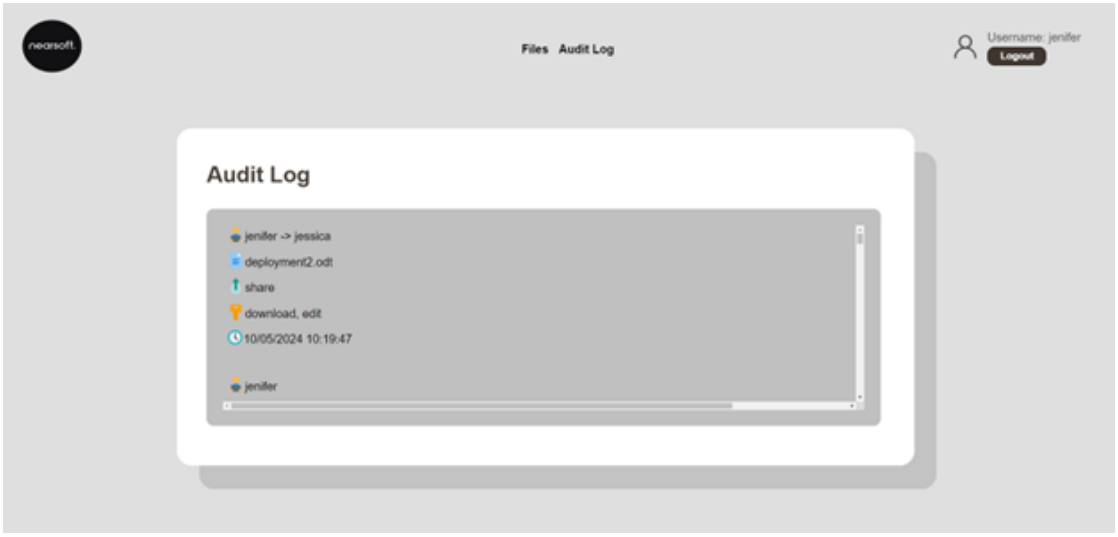


Figure G.16: Share File - audit log

Figure G.17: Share File - jessica page

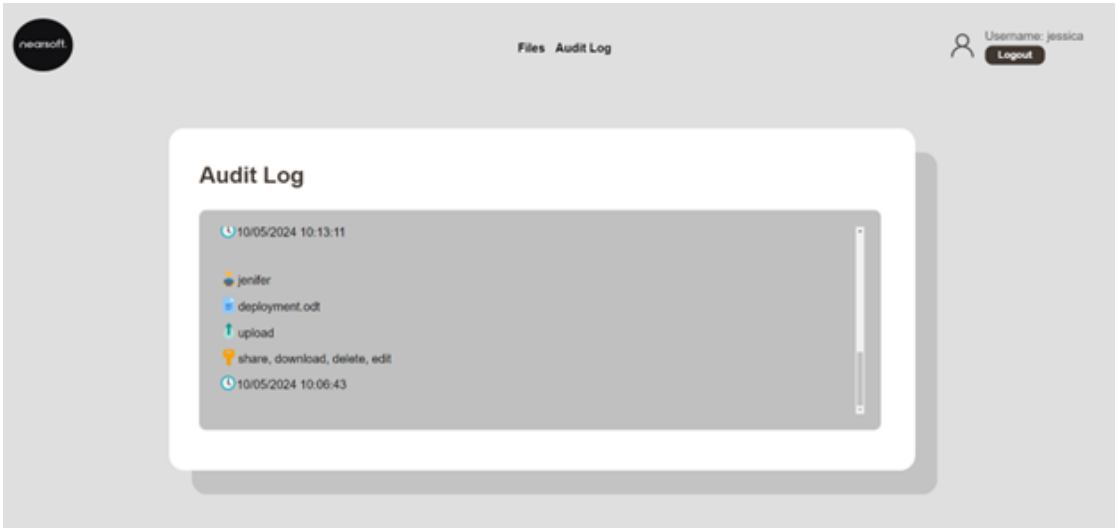


Figure G.18: Share File - audit log jessica page

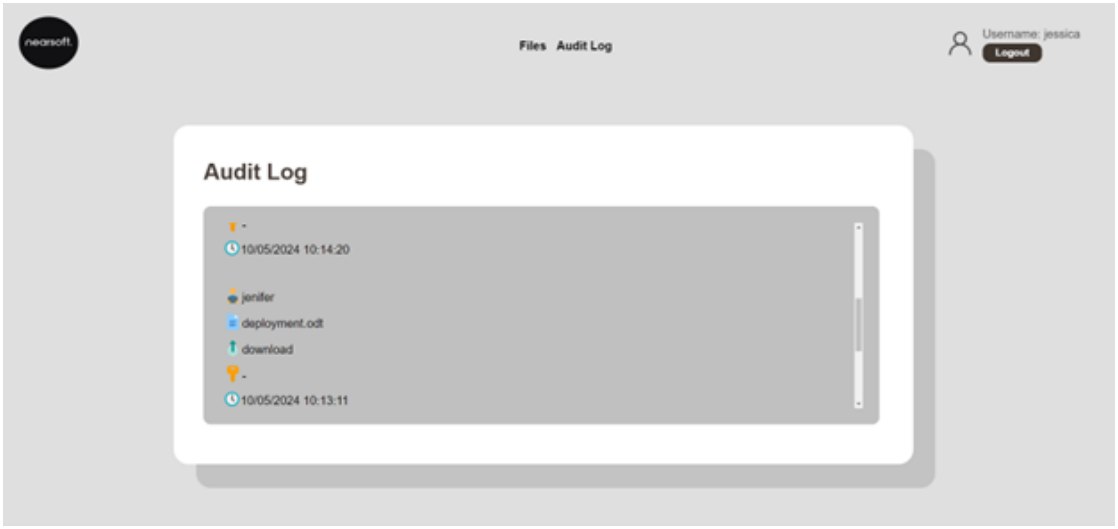


Figure G.19: Share File - audit log jessica page (cont.)

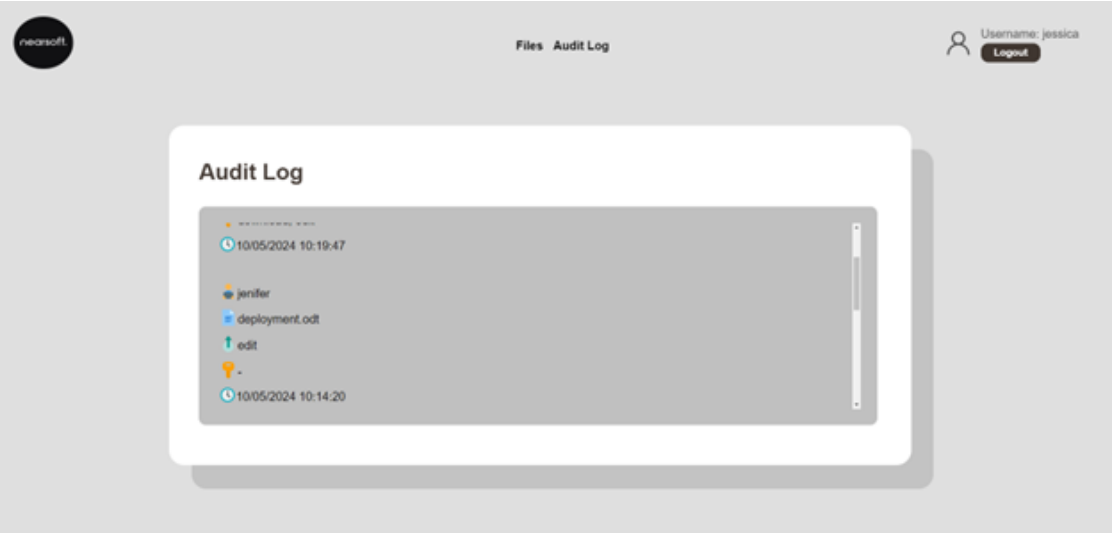


Figure G.20: Share File - audit log jessica page (cont.)

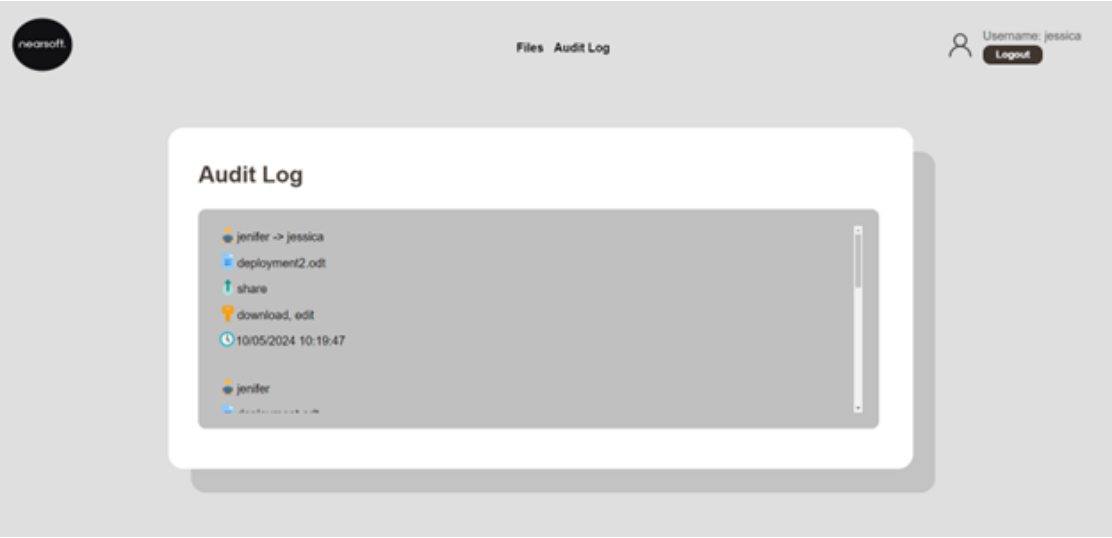


Figure G.21: Share File - audit log jessica page (cont.)

G.3.5 Delete File

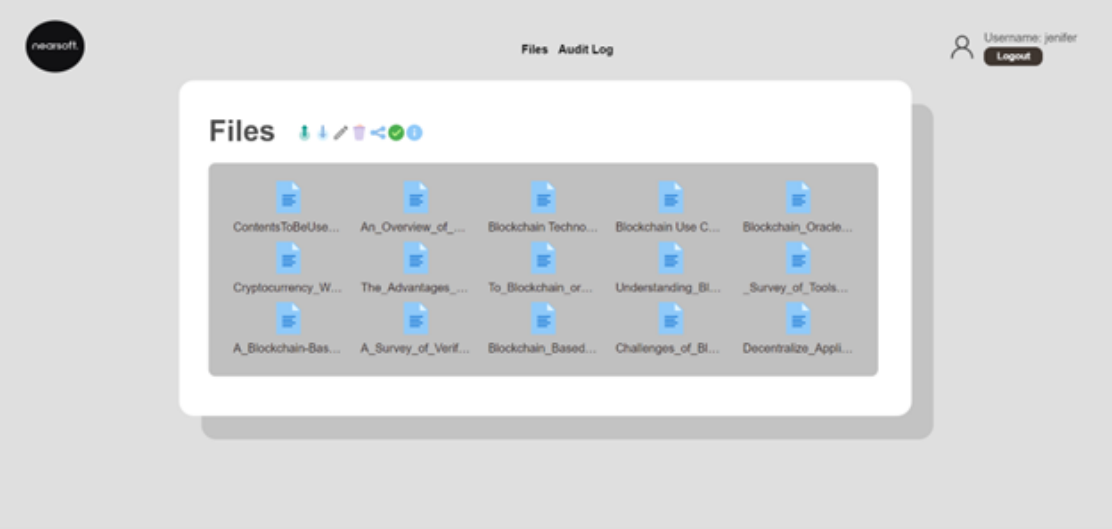


Figure G.22: Delete File - File view

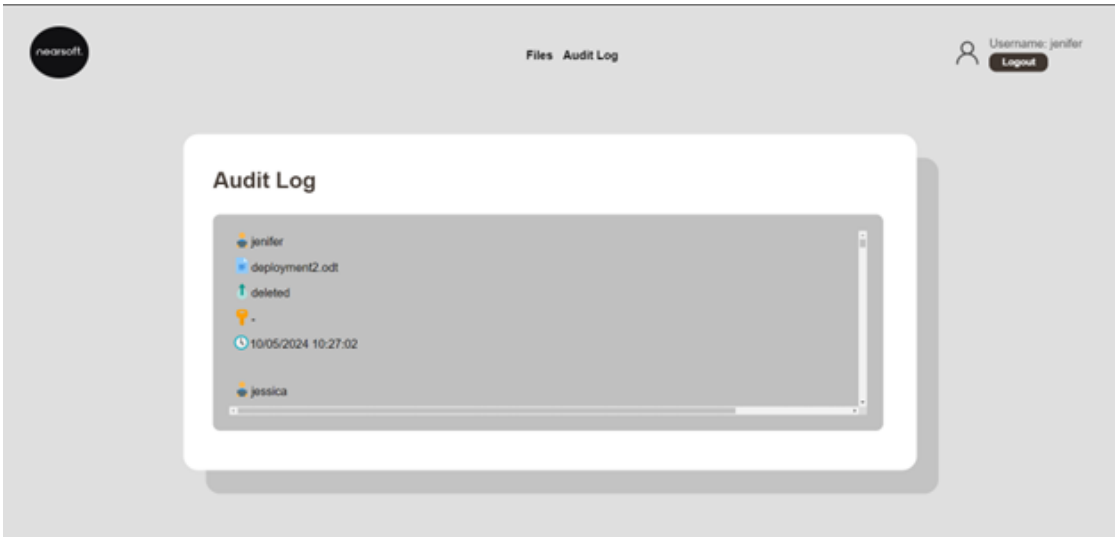


Figure G.23: Delete File - Audit Log

G.3.6 Download File

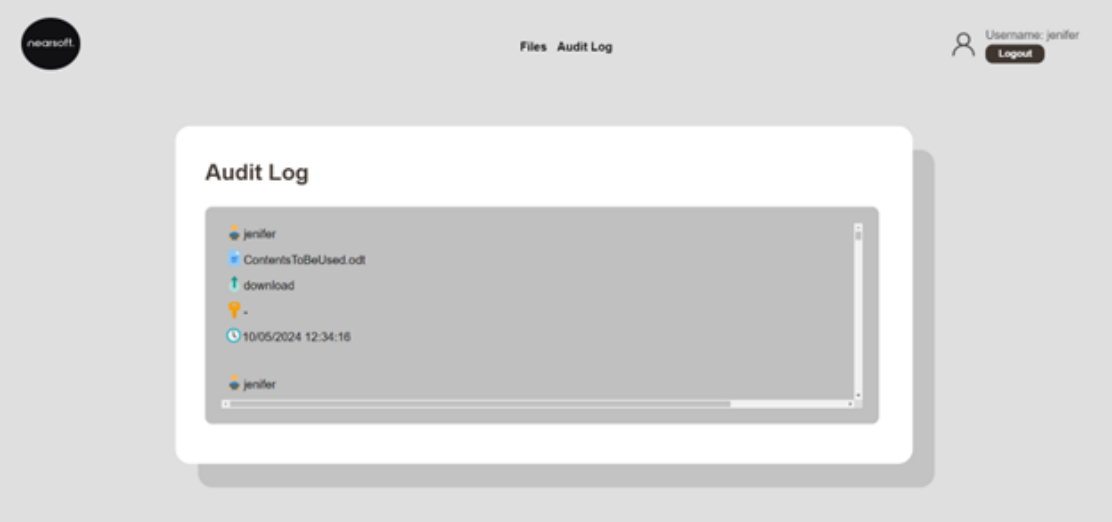


Figure G.24: Download File - Audit Log

G.3.7 Verify File

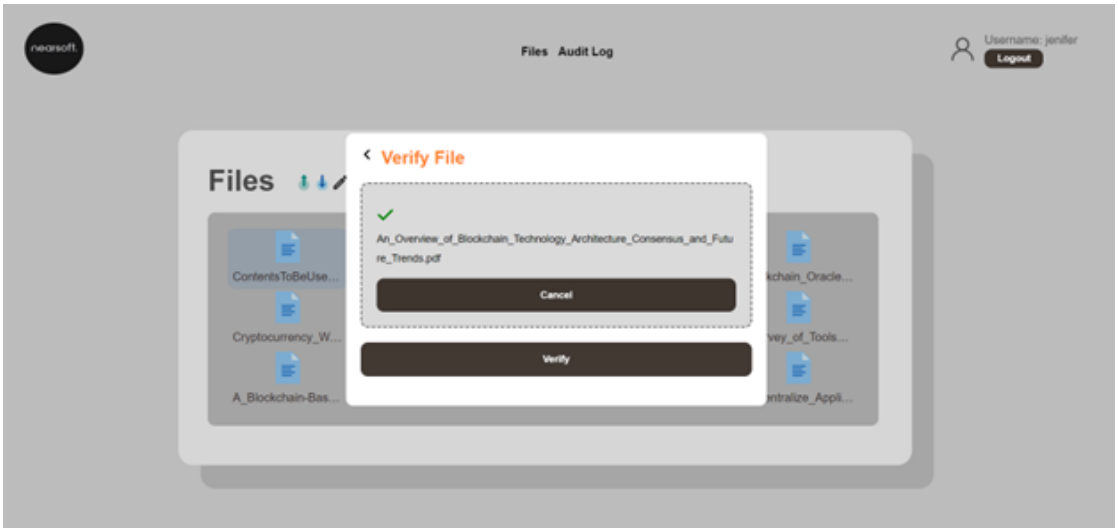


Figure G.25: Verify File

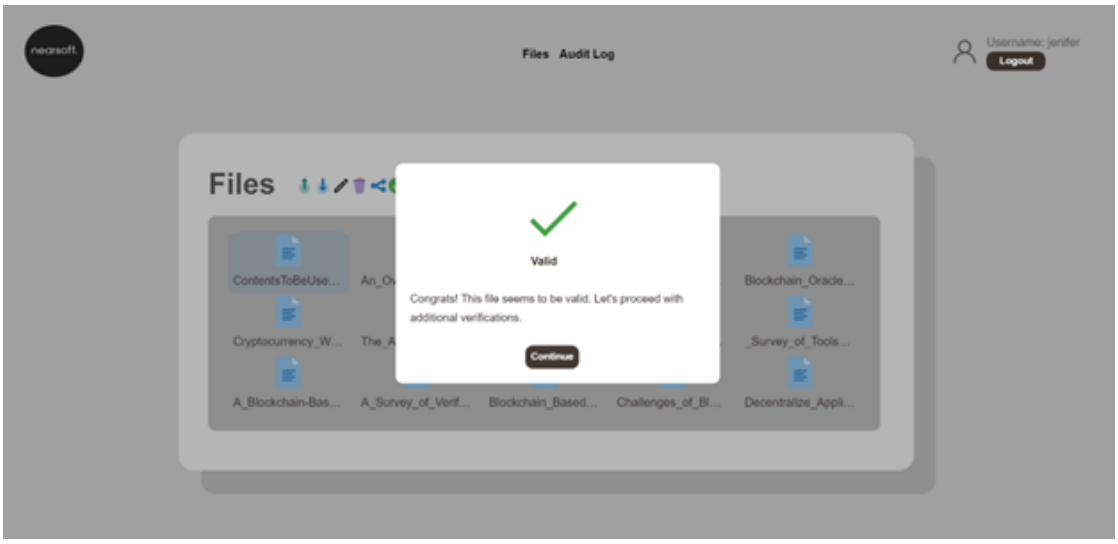


Figure G.26: Verify File - Valid File

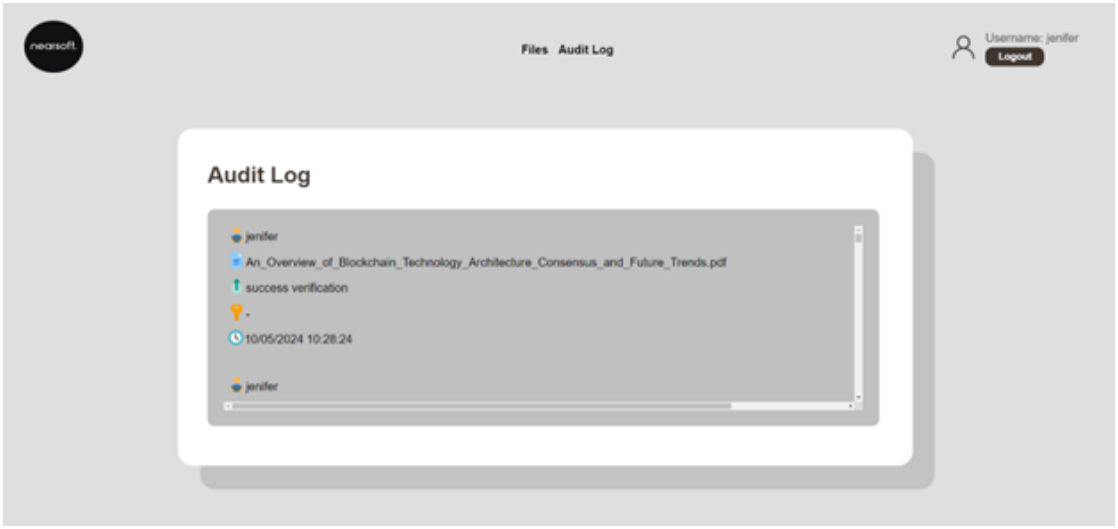


Figure G.27: Verify File - Audit Log

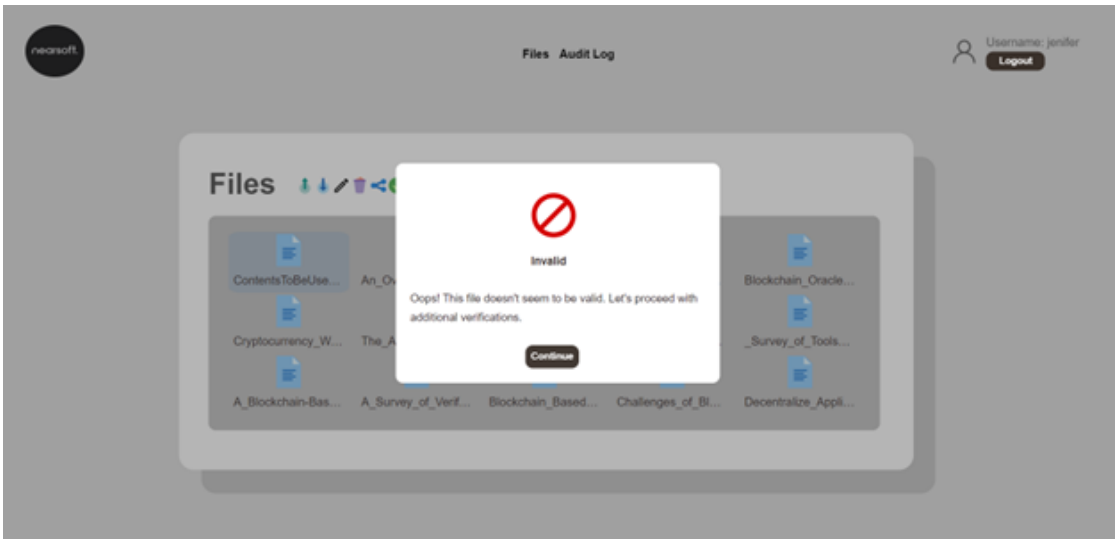


Figure G.28: Verify File - Invalid File

Appendix H

Unit Testing

This appendix shows the results of the coverage testing tools both in the front end and back end.

H.1 Front end

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	94.06	94.44	91.17	93.96	
components	95.45	100	85.71	95.45	
SessionExpirationHandler.jsx	95.45	100	85.71	95.45	12
helpers	75	100	75	75	
UserApp.js	75	100	75	75	13-18,61-62
helpers/Commands	96.77	93.33	95.65	96.7	
Command.js	0	0	0	0	
DeleteFileCommand.js	100	100	100	100	
DownloadFileCommand.js	100	100	100	100	
DropEdit.js	100	100	100	100	
DropUpload.js	100	100	100	100	
ShareFileCommand.js	94.11	87.5	100	93.93	33-34
UpdatePermissionsCommand.js	94.59	90	100	94.28	65,72
VerifyFileCommand.js	100	100	100	100	

```
Test Suites: 10 passed, 10 total
Tests:       30 passed, 30 total
Snapshots:   0 total
Time:        5.797 s
Ran all test suites.
```

Figure H.1: Test Coverage in Front end using Jest

H.2 Back end

```
  121 passing (40s)

----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts\           |    98.39 |    91.57 |    98.57 |    98.08 |                |
  AccessControl.sol   |    99.26 |    92.86 |      100 |    99.36 |            515 |
  AuditLogControl.sol |    94.12 |    83.33 |      100 |     87.5 |       69,70,71 |
  FileRegister.sol    |    97.22 |    80.77 |      100 |    98.25 |            180 |
  Helper.sol          |      100 |      100 |      100 |      100 |                |
  LoginRegister.sol   |      100 |      100 |      100 |      100 |                |
  UserRegister.sol    |    96.43 |    95.83 |    91.67 |    96.97 |            126 |
----------------------|----------|----------|----------|----------|----------------|
All files             |    98.39 |    91.57 |    98.57 |    98.08 |                |
----------------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
```

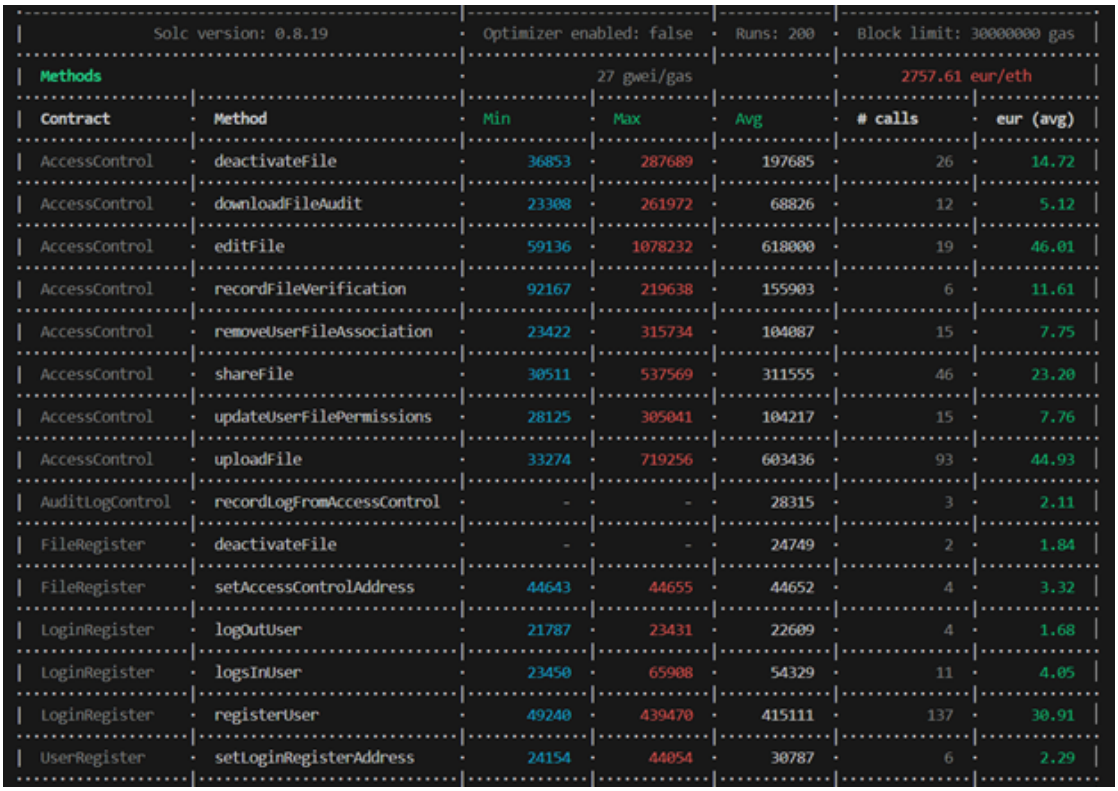
Figure H.2: Test Coverage in Backend end using HardHat Testing Environment

H.3 Eth-Gas-Reporter

```
require("@nomicfoundation/hardhat-toolbox");
require('dotenv').config();  6.4k (gzipped: 2.8k)

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.19",
  paths: {
    artifacts: '../client/src/contracts',
  },
  networks: {
    hardhat: {
      allowUnlimitedContractSize: true,
      chainId: 31337,
    }
  },
  gasReporter: {
    currency: 'EUR', // currency which gas costs are reported
    gasPrice: 27,    // gas price to be used for calculating gas costs - adjust according to the fas price in the network
    enabled: true,
    coinmarketcap: process.env.COINMARKETCAP_API_KEY,
    // --- To create a report file uncomment these lines bellow ---
    //outputFile: 'gar-report.txt', // gar report file name
    //noColors: true,
  },
};
```

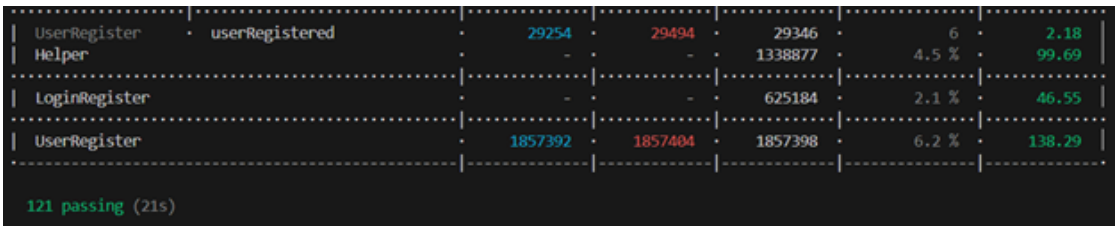
Figure H.3: HardHat Configuration for eth-gas-reporter



Solc version: 0.8.19	Optimizer enabled: false	Runs: 200	Block limit: 30000000 gas
Methods	27 gwei/gas		2757.61 eur/eth

Contract	Method	Min	Max	Avg	# calls	eur (avg)
AccessControl	deactivateFile	36853	287689	197685	26	14.72
AccessControl	downloadFileAudit	23308	261972	68826	12	5.12
AccessControl	editFile	59136	1078232	618000	19	46.01
AccessControl	recordFileVerification	92167	219638	155903	6	11.61
AccessControl	removeUserFileAssociation	23422	315734	104087	15	7.75
AccessControl	shareFile	30511	537569	311555	46	23.20
AccessControl	updateUserFilePermissions	28125	305041	104217	15	7.76
AccessControl	uploadFile	33274	719256	603436	93	44.93
AuditLogControl	recordLogFromAccessControl	-	-	28315	3	2.11
FileRegister	deactivateFile	-	-	24749	2	1.84
FileRegister	setAccessControlAddress	44643	44655	44652	4	3.32
LoginRegister	logOutUser	21787	23431	22609	4	1.68
LoginRegister	logsInUser	23450	65908	54329	11	4.05
LoginRegister	registerUser	49240	439470	415111	137	30.91
UserRegister	setLoginRegisterAddress	24154	44054	30787	6	2.29

Figure H.4: Eth-Gas-Reporter



UserRegister	userRegistered	29254	29494	29346	6	2.18
Helper		-	-	1338877	4.5 %	99.69
LoginRegister		-	-	625184	2.1 %	46.55
UserRegister		1857392	1857404	1857398	6.2 %	138.29

121 passing (21s)

Figure H.5: Eth-Gas-Reporter

Appendix I

Security Coverage

This appendix shows the results of the security coverage tools on the front and back end.

I.1 Front end

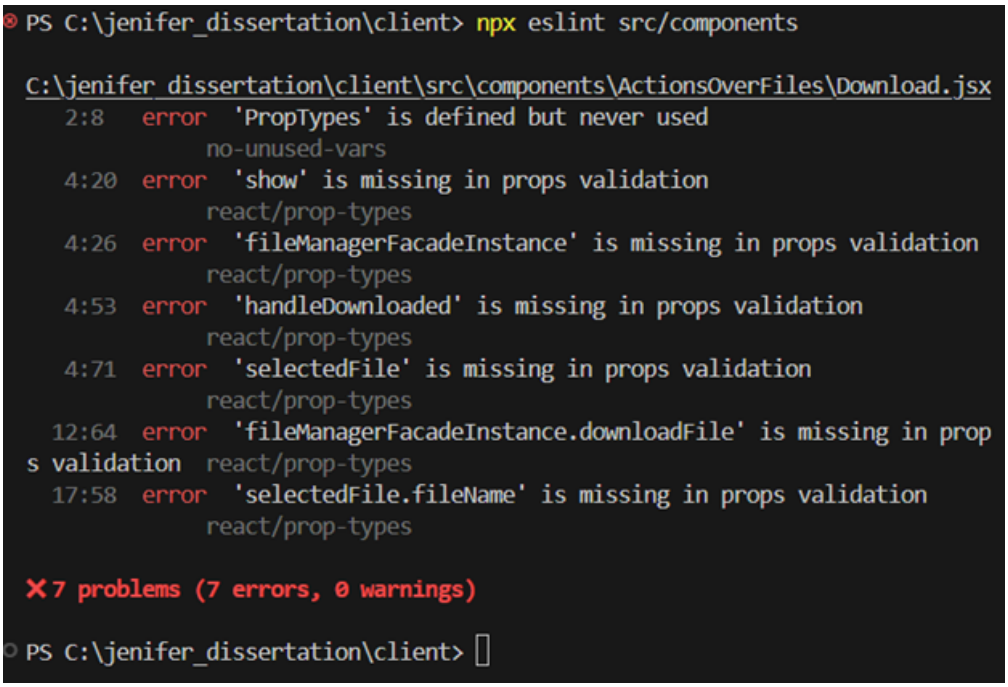


Figure I.1: EsLint problem.

README.md M | hardhat.config.js | Download.jsx

client > src > components > ActionsOverFiles > Download.jsx > ...

```
4   const Download = ({show, fileManagerFacadeInstance, handleDownloaded, selectedFile}) => {
7       useEffect(() => {
8           const handleDownload = async () => {
10                  console.log('Download file ...');
11                  try {
12                      var blob = await fileManagerFacadeInstance.downloadFile(selectedFile);
13
14                      // Creates a downloaded link
15                      const downloadLink = document.createElement("a");
16                      downloadLink.href = URL.createObjectURL(blob);
17                      downloadLink.download = selectedFile.fileName;
18                      document.body.appendChild(downloadLink);
19                      downloadLink.click();
20                      document.body.removeChild(downloadLink);
21
22                      handleDownloaded();
23                  } catch (error) {
24                      console.error('Error downloading file:', error);
25                  }
26              }
27          }
28
29          handleDownload();
30      }, [fileManagerFacadeInstance, handleDownloaded, selectedFile, show]);
31
32      return null;
33  }
34
35  Download.propTypes = {      You, last month • ESLint fixes
36      show: PropTypes.bool.isRequired,
37      fileManagerFacadeInstance: PropTypes.object.isRequired,
38      handleDownloaded: PropTypes.func.isRequired,
39      selectedFile: PropTypes.object
40  };
41
42  export default Download;
```

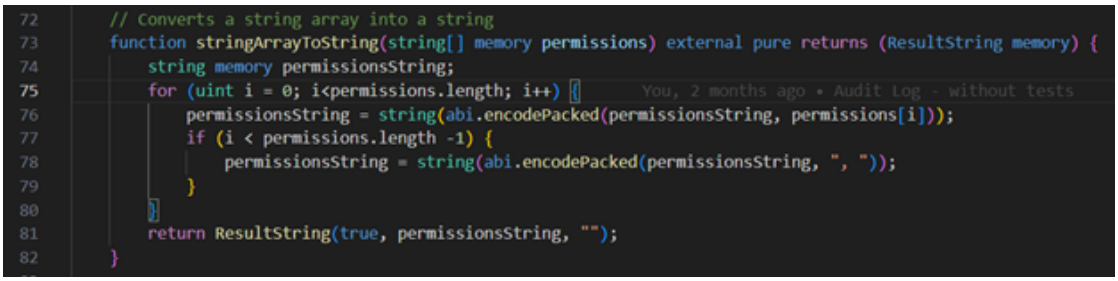
Figure I.2: Solution to the EsLint problem

I.2 Back end

I.2.1 Slither Error/Solution

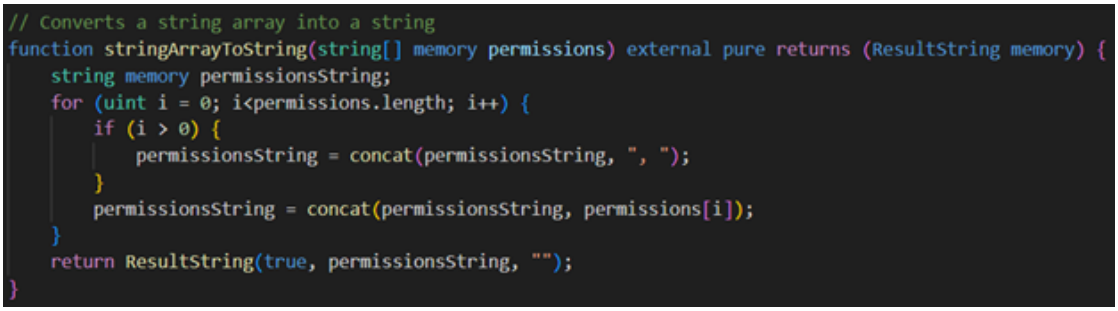
```
Helper.stringArrayToString(string[]) (contracts/Helper.sol#73-82) calls abi.encodePacked() with multiple dynamic arguments:
        - permissionsString = string(abi.encodePacked(permissionsString,permissions[i])) (contracts/Helper.sol#76)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#abi-encodePacked-collision
```

Figure I.3: Problem: abi.encodedPacked() with multiple dynamic arguments.



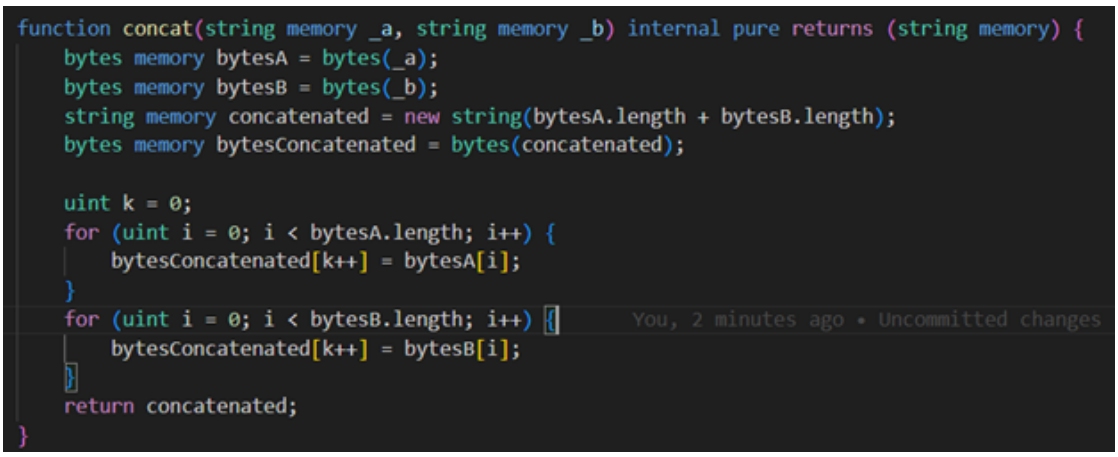
```
72      // Converts a string array into a string
73      function stringArrayToString(string[] memory permissions) external pure returns (ResultString memory) {
74          string memory permissionsString;
75          for (uint i = 0; i<permissions.length; i++) {        You, 2 months ago • Audit Log - without tests
76              permissionsString = string(abi.encodePacked(permissionsString, permissions[i]));
77              if (i < permissions.length -1) {
78                  permissionsString = string(abi.encodePacked(permissionsString, ", "));
79              }
80          }
81          return ResultString(true, permissionsString, "");
82      }
```

Figure I.4: Code that lead to the problem: abi.encodedPacked() with multiple dynamic arguments.



```
// Converts a string array into a string
function stringArrayToString(string[] memory permissions) external pure returns (ResultString memory) {
    string memory permissionsString;
    for (uint i = 0; i<permissions.length; i++) {
        if (i > 0) {
            permissionsString = concat(permissionsString, ", ");
        }
        permissionsString = concat(permissionsString, permissions[i]);
    }
    return ResultString(true, permissionsString, "");
}
```

Figure I.5: Solution: abi.encodedPacked() with multiple dynamic arguments.



```
function concat(string memory _a, string memory _b) internal pure returns (string memory) {
    bytes memory bytesA = bytes(_a);
    bytes memory bytesB = bytes(_b);
    string memory concatenated = new string(bytesA.length + bytesB.length);
    bytes memory bytesConcatenated = bytes(concatenated);

    uint k = 0;
    for (uint i = 0; i < bytesA.length; i++) {
        bytesConcatenated[k++] = bytesA[i];
    }
    for (uint i = 0; i < bytesB.length; i++) {        You, 2 minutes ago • Uncommitted changes
        bytesConcatenated[k++] = bytesB[i];
    }
    return concatenated;
}
```

Figure I.6: Solution: abi.encodedPacked() with multiple dynamic arguments.

I.2.2 Slither Strict Equality Warning

```
AuditLogControl.getLogs(string[]) (contracts/AuditLogControl.sol#50-87) uses a dangerous strict equality:
        - keccak256(bytes)(abi.encodePacked(logs[j].fileIpfsCid)) == fileHash (contracts/AuditLogControl.sol#57)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
```

Figure I.7: Warning: Strict Equality Warning.

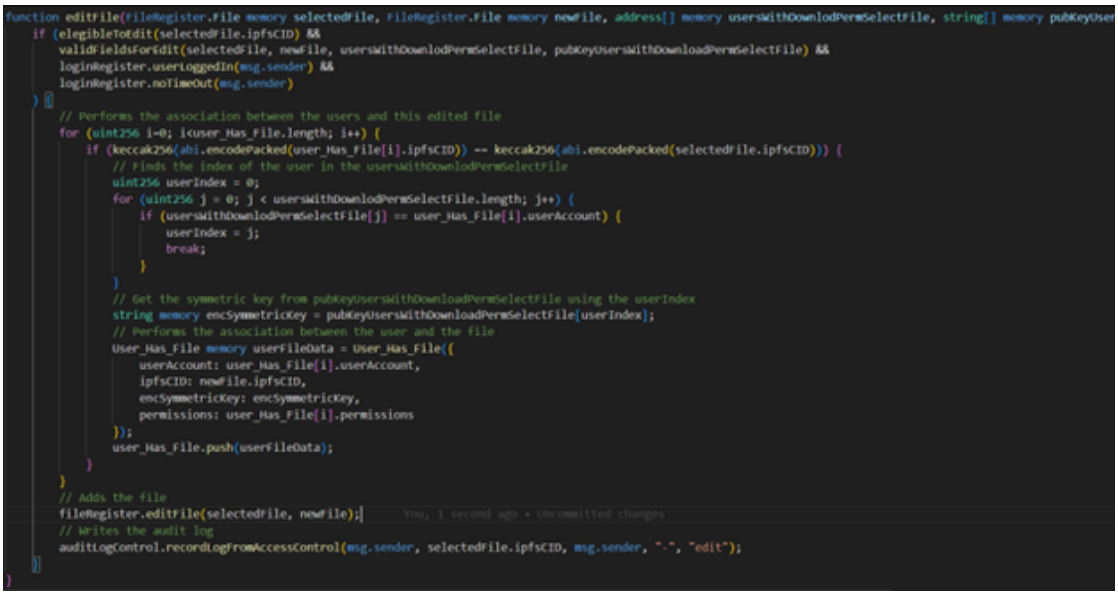
I.2.3 Slither Reentrancy Attack Warning

```
Reentrancy in AccessControl.editFile(FileRegister.File,FileRegister.File,address[],string[]) (contracts/AccessControl.sol#78-112):
        External calls:
        - fileRegister.editFile(selectedFile,newFile) (contracts/AccessControl.sol#85)
        State variables written after the call(s):
        - user_Has_File.push(userFileData) (contracts/AccessControl.sol#106)
        AccessControl.user_Has_File (contracts/AccessControl.sol#18) can be used in cross function reentrancies:
        - AccessControl.editFile(FileRegister.File,FileRegister.File,address[],string[]) (contracts/AccessControl.sol#78-112)
        - AccessControl.getEncSymmetricKeyFileUser(address,string) (contracts/AccessControl.sol#281-294)
        - AccessControl.getPermissionsOverFile(address,string) (contracts/AccessControl.sol#323-334)
        - AccessControl.getUserFiles(address,string) (contracts/AccessControl.sol#361-387)
        - AccessControl.getUsersAssociatedWithFile(string) (contracts/AccessControl.sol#337-358)
        - AccessControl.messageSenderAssociatedToFile(string) (contracts/AccessControl.sol#436-447)
        - AccessControl.recordFileVerification(address,string) (contracts/AccessControl.sol#199-215)
        - AccessControl.removeUserFileAssociation(address,string) (contracts/AccessControl.sol#255-275)
        - AccessControl.shareFile(address,string,string[],string[]) (contracts/AccessControl.sol#123-150)
        - AccessControl.updateUserFilePermissions(address,string,string[]) (contracts/AccessControl.sol#159-176)
        - AccessControl.uploadFile(address,FileRegister.File,string) (contracts/AccessControl.sol#46-71)
        - AccessControl.userAssociatedWithFile(address,string) (contracts/AccessControl.sol#404-416)
        - AccessControl.userAssociatedWithFileName(address,string) (contracts/AccessControl.sol#420-433)
        - AccessControl.verifyValidFile(address,string) (contracts/AccessControl.sol#221-234)
```

Figure I.8: Warning: Reentrancy Warning.

```
function editFile(FileRegister.File memory selectedFile, FileRegister.File memory newFile, address[] memory userswithDownlodPermSelectFile, string[] memory pubKeyUse
    if (elegibleToEdit(selectedFile.ipfsCID) &&
        validFieldsForEdit(selectedFile, newFile, userswithDownlodPermSelectFile, pubKeyUserswithDownloadPermSelectFile) &&      You, 4 weeks ago • added validatio
        loginRegister.userLoggedIn(msg.sender) &&
        loginRegister.noTimeOut(msg.sender)
    ) {
        // Adds the file
        fileRegister.editFile(selectedFile, newFile);
        // Performs the association between the users and this edited file
        for (uint256 i=0; i<user_Has_File.length; i++) {
            if (keccak256(abi.encodePacked(user_Has_File[i].ipfsCID)) == keccak256(abi.encodePacked(selectedFile.ipfsCID))) {
                // Finds the index of the user in the userswithDownlodPermSelectFile
                uint256 userIndex = 0;
                for (uint256 j = 0; j < userswithDownlodPermSelectFile.length; j++) {
                    if (userswithDownlodPermSelectFile[j] == user_Has_File[i].userAccount) {
                        userIndex = j;
                        break;
                    }
                }
                // Get the symmetric key from pubKeyUserswithDownloadPermSelectFile using the userIndex
                string memory encSymmetricKey = pubKeyUserswithDownloadPermSelectFile[userIndex];
                // Performs the association between the user and the file
                User_Has_File memory userFileData = User_Has_File({
                    userAccount: user_Has_File[i].userAccount,
                    ipfsCID: newFile.ipfsCID,
                    encSymmetricKey: encSymmetricKey,
                    permissions: user_Has_File[i].permissions
                });
                user_Has_File.push(userFileData);
            }
        }
        // Writes the audit log
        auditLogControl.recordLogFromAccessControl(msg.sender, selectedFile.ipfsCID, msg.sender, "-", "edit");
    }
}
```

Figure I.9: Code that originated the warning for the Reentrancy Attack.



```
function editFile(FileRegister.File memory selectedFile, FileRegister.File memory newFile, address[] memory usersWithDownlodPermSelectFile, string[] memory pubKeyUser
    if (elegibleToEdit(selectedFile.ipfsCID) &&
        validFieldsForEdit(selectedFile, newFile, usersWithDownlodPermSelectFile, pubKeyUsersWithDownloadPermSelectFile) &&
        loginRegister.userLoggedIn(msg.sender) &&
        loginRegister.noTimeOut(msg.sender)
    ) {
        // Performs the association between the users and this edited file
        for (uint256 i=0; i<user_Has_File.length; i++) {
            if (keccak256(abi.encodePacked(user_Has_File[i].ipfsCID)) == keccak256(abi.encodePacked(selectedFile.ipfsCID))) {
                // Finds the index of the user in the usersWithDownlodPermSelectFile
                uint256 userIndex = 0;
                for (uint256 j = 0; j < usersWithDownlodPermSelectFile.length; j++) {
                    if (usersWithDownlodPermSelectFile[j] == user_Has_File[i].userAccount) {
                        userIndex = j;
                        break;
                    }
                }
                // Get the symmetric key from pubKeyUsersWithDownloadPermSelectFile using the userIndex
                string memory encSymmetricKey = pubKeyUsersWithDownloadPermSelectFile[userIndex];
                // Performs the association between the user and the file
                User_Has_File memory userFileData = User_Has_File({
                    userAccount: user_Has_File[i].userAccount,
                    ipfsCID: newFile.ipfsCID,
                    encSymmetricKey: encSymmetricKey,
                    permissions: user_Has_File[i].permissions
                });
                user_Has_File.push(userFileData);
            }
        }
        // Adds the file
        fileRegister.editFile(selectedFile, newFile);      You, 1 second ago • Uncommitted changes
        // Writes the audit log
        auditLogControl.recordLogFromAccessControl(msg.sender, selectedFile.ipfsCID, msg.sender, "-", "edit");
    }
}
```

Figure I.10: Solution for the Reentrancy Attack Warning.