

MINISTRY OF EDUCATION



TECHNICAL UNIVERSITY

OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE

SUPPLY CHAIN MANAGEMENT USING DISTRIBUTED LEDGERS FOR DATA IMMUTABILITY AND INTEGRITY

LICENSE THESIS

Graduate: **Georgiana-Grațîela STÂNEA**

Supervisor: **Șl. dr. ing. Marcel ANTAL**

2024

Contents

Chapter 1 Introduction	1
1.1 Project Context	1
1.2 Thesis Structure	2
Chapter 2 Project Objectives	3
2.1 Main Objective	3
2.2 Secondary Objectives	3
2.3 Requirements	4
2.3.1 Functional Requirements	4
2.3.2 Non-Functional Requirements	4
2.4 Metrics used for evaluation	5
Chapter 3 Bibliographic Research	7
3.1 Blockchain	7
3.1.1 Bitcoin	8
3.1.2 Ethereum	8
3.2 Data Storage	9
3.3 Alternative Anti-Counterfeit Technologies	10
3.3.1 QR Codes	10
3.3.2 Radio-Frequency Identification tags	10
3.3.3 AI-Based Technologies	11
Chapter 4 Analysis and Theoretical Foundation	12
4.1 Ethereum Blockchain	12
4.1.1 Keys and Addresses	12
4.1.2 Transactions	14
4.1.3 Consensus	15
4.1.4 Wallets	16
4.1.5 Smart Contracts	17
4.2 InterPlanetary File System	18
4.3 QR Codes	20
4.4 Use Cases and Scenarios	21
4.4.1 Use Case 1: Manufacturer adds a product	21
4.4.2 Use Case 2: Supplier updates a product	22
4.4.3 Use Case 3: Customer searches for a product	23
Chapter 5 Detailed Design and Implementation	25
5.1 System Design	25
5.1.1 Web Application	26
5.1.2 Ethereum Blockchain	27
5.1.3 IPFS	28
5.2 System Implementation	28
5.2.1 Smart Contract	28
5.2.2 Smart Contract Deployment	32
5.2.3 Interactions with the Blockchain	33

5.2.4 Interactions with IPFS	39
5.2.5 QR Code Generation	40
Chapter 6 Testing and Validation	41
6.1 Experimental Setup	41
6.2 Experimental Results	42
6.2.1 Storing files on the Blockchain	42
6.2.2 Storing files using IPFS	43
6.2.3 Analysis	44
Chapter 7 User's Manual	46
7.1 Installation Prerequisites	46
7.2 Installation Guideline	46
7.3 User Interaction	46
7.3.1 Manufacturer	47
7.3.2 Supplier	49
7.3.3 Customer	50
Chapter 8 Conclusions	52
8.1 Accomplished objectives	52
8.2 Personal contributions	53
8.3 Future improvements	53
Bibliography	55
Appendix A <i>ProductTracker</i> Smart Contract	58
Appendix B <i>ProductTracker</i> Smart Contract without IPFS	60
Appendix C Published Papers	64

Chapter 1. Introduction

Counterfeit goods are a widespread issue affecting numerous sectors, from electronics and automotive components to consumer products and pharmaceuticals. This problem results in manufacturers and suppliers losing billions of dollars each year. However, the risks consumers are exposed to are often more severe. Fake auto parts or consumer goods that malfunction can lead to overheating or fires, and counterfeit medications contribute to over a million deaths annually.

1.1. Project Context

In the domain of supply chain management, the detection and prevention of counterfeit products represent a significant and complex challenge. The lack of transparency in traditional supply chains led to issues such as service redundancy, poor coordination among departments, and lack of standardization, contributing to the extension of counterfeit goods.

The counterfeit cosmetics market worldwide has been experiencing a notable increase, mirroring global trends in the widespread distribution of fake makeup products. According to a report by the European Union Intellectual Property Office (EUIPO) [1], the cosmetics sector in the EU loses approximately €3 billion annually due to the presence of counterfeit products. This figure represents a significant portion of the market, with counterfeit goods not only affecting consumer confidence but also causing substantial financial losses for genuine brands.

The impact of using counterfeit cosmetics can be severe, producing health risks due to the presence of hazardous substances. European authorities have reported instances of counterfeit products containing dangerous levels of heavy metals such as lead and mercury, alongside carcinogenic substances like arsenic and cadmium. These findings underscore the health hazards posed by such products. The European Commission's rapid alert system for dangerous non-food products has consistently highlighted the issue, with numerous alerts issued for counterfeit cosmetics that fail to comply with EU health and safety standards [2].

Challenges also arise for legitimate manufacturers, impacting them in various ways including reduced sales, the expenses associated with brand protection, bad reputation, and the financial burden of disposing counterfeit goods and legal actions against counterfeiters or individuals affected by fake products. These issues are often referenced in a broad sense within corporate documentation. For instance, Pfizer, one of the top five global pharmaceutical firms, acknowledged the issue of counterfeiting in its 2019 annual financial statement. Within this financial document, the company dedicates a section to counterfeit products [3], offering an overview of the challenges encountered and detailing the measures undertaken to combat these issues.

Customers require credible ways of verifying the authenticity of the products they

are interested in without being exposed to the unwanted side-effects. The same applies to brands and manufacturers. The proposed system effectively addresses these challenges by incorporating blockchain technology to enhance the transparency and trust across the entire supply chain, ensuring that all parties involved have access to reliable information regarding product authenticity.

1.2. Thesis Structure

This section provides a brief overview of each chapter contained within this thesis document.

Chapter 1 serves as an introduction, outlining the context, motivation, and a brief description of the objectives this project aims to achieve. The second part of this chapter outlines the structure of the work.

Chapter 2 provides a detailed presentation of the work's objectives, divided into main and secondary objectives, as well as the requirements that need to be met. These requirements fall into two categories: functional and non-functional. Both the requirements and objectives will be measured using metrics described in this section.

Chapter 3 presents relevant literature concerning the selected thesis topic and various methodologies for integrating blockchain technology within the supply chain domain.

Chapter 4 discusses and theoretically analyzes the solutions implemented in the proposed system, providing justification for each component of its structure.

Chapter 5 outlines the primary components of the system and presents their interactions. It provides a detailed overview of the project's implementation by describing the structure of the application.

Testing and validation of the solution will be conducted in Chapter 6, where there are specified the methods used for testing the application and it is presented a comparison between two approaches.

Chapter 7 provides a guide outlining the essential steps users need to follow to execute and utilize the application. The workflow of the application is presented for all types of users.

The final chapter summarizes the work, the objectives achieved, and completed, as well as potential improvements that could be added later to the proposed system.

Chapter 2. Project Objectives

2.1. Main Objective

The main objective of the thesis is to study, design and develop a decentralized application that ensures the authenticity and traceability of products by allowing end users and suppliers to track a product's entire supply chain history. Through this approach, the application aims to build trust between all parties involved, significantly lowering the incidence of counterfeiting in the marketplace.

2.2. Secondary Objectives

Table 2.1: Thesis secondary objectives

No.	Objective	Description	Reference
1	Study of the existing counterfeit detection methods.	Several proposed solutions and systems were studied and analyzed.	Ch. 3
2	Study of the specialized literature in the field of blockchain supply chain applications.	Understanding the concepts of blockchain.	Ch. 4
3	Study of the specialized literature regarding distributed data storage.	Comparing distributed data storage technologies. A single method will be chosen to be introduced in the system implementation.	Ch. 3
4	Study of the specialized literature regarding data encoding and decoding using cryptography.	Analysis of symmetric and asymmetric encryption methods.	Ch. 4
5	Implement a solution that assures transparency, security, and efficiency in supply chain management.	The implementation will focus on integrating blockchain technology to ensure an unalterable record of transactions.	Ch. 5

2.3. Requirements

2.3.1. Functional Requirements

The system is designed for supporting three types of users: manufacturers, suppliers and customers. The functional requirements for these roles are the following:

- A user can select the network on which they want to make their transactions.
- Manufacturers and suppliers have to connect to their Ethereum accounts using the Metamask Wallet.
- After the connection is established, a manufacturer can perform the following actions:
 - View the address of its own account.
 - Register a new product on the blockchain by entering comprehensive details about it.
 - Approve or revoke the transaction associated with registering a product on the blockchain.
 - Generate the QR code corresponding to the added product.
 - Download the QR code on the local device.
 - Verify the details of the completed action on Etherscan, after a transaction is successful.
- A connected supplier can perform the following actions:
 - View the address of its own account.
 - Scan a QR code associated with a product by turning his device camera on.
 - Upload a QR code from his local device.
 - Updating an existing product on the blockchain by entering the new details about it.
 - Register a bundle composed of products that already exist on the blockchain.
 - Approve or revoke the transaction associated with updating or adding a product on the blockchain.
 - Generate QR codes for products.
 - Download the QR code on the local device.
 - Verify the details of the completed action on Etherscan, after a transaction is successful.
- A customer can perform the following actions:
 - Search a product by its serial number.
 - Scan a QR code associated with a product by turning his device camera on.
 - Upload a QR code from his local device.
 - View the history of the product he is interested in.

2.3.2. Non-Functional Requirements

1. Security

- The information about a product is stored on IPFS, which is a distributed file system that uses asymmetric encryption for handling data.
- Blockchain employs a consensus mechanism that assures all the operations made on products are validated by the entire network.
- Smart contracts enhance their security through Solidity's built-in function.

2. Immutability

- Information about products have to stay unchanged to prevent counterfeiters from setting a favorable history.
 - The blockchain ensures that once data is entered, it cannot be altered or deleted, guaranteeing the integrity of the supply chain.
3. Scalability
 - This non-functional requirement is achieved by using IPFS, a distributed file system.
 - IPFS offers better performance than blockchain for storing data, especially in the case of product bundles.
 4. Availability
 - The presented system should be available anytime, from any location and across all time zones, ensuring users can access it regardless of their geographical position.
 - As a result of using blockchain technology, data is replicated across multiple nodes, keeping the system operational even if parts of the network fail.
 5. Transparency
 - This non-functional requirement refers to the quality of the system to be accessible in the same manner for all the parties involved.
 - The operations and information about a product are visible to all users, creating a transparent environment where one can track product histories and transaction details.
 6. Usability
 - This non-functional requirement points out how good the user's experience is.
 - The system uses QR codes for product tracking, which offers an easy solution for verifying product history through a simple scan.
 - The system's usability is also improved by displaying error or success messages based on user's actions.
 7. Extensibility
 - The system must be designed such that it allows for future developments, expansions, or changes, depending on the scenarios used and the requirements that need to be met.

2.4. Metrics used for evaluation

Integrating a system that identifies counterfeit products onto the blockchain represents significant challenges, particularly in terms of cost and scalability. The process involves frequent updates and transactions on the blockchain to record and verify product authenticity. However, the cumulative number of transactions required for this integration can result in substantial expenses due to gas fees and other associated costs.

One critical metric for evaluating these aspects is the number of transactions recorded per second (TPS), which represents the number of transactions that the network can process within a second. TPS is a fundamental indicator of the network's capacity to handle transactional demands effectively and efficiently.

A higher TPS indicates that the blockchain network can accommodate a larger volume of transactions, thereby supporting a more significant number of users and applications. On the other hand, a lower TPS suggests limitations in the network's scalability and may lead to transaction backlogs, delays, and increased transaction costs during periods of high activity.

The throughput of blockchain can be calculated using the formula:

$$TPS = \frac{block_gas}{transaction_gas} \times \frac{1}{block_time}$$

The block gas represents the maximum amount of gas that can be consumed in a single block. Gas is a unit of measurement for the computational work performed on the Ethereum network. Each operation executed by the Ethereum Virtual Machine (EVM) consumes a specific amount of gas. According to a chart [4] provided by Etherscan, the current gas limit is approximately 30,000,000.

The current block time for Ethereum is approximately 12 seconds [5]. The minimum amount paid for every transaction on Ethereum is 21,000 gas, according to Ethereum's Yellow Paper [6].

Considering these, the formula becomes:

$$\frac{30000000}{21000} \times \frac{1}{12} \approx 119 \text{ TPS}$$

Storing files on a blockchain, such as Ethereum, incurs fees similar to any other blockchain transaction. When considering the gas costs for adding or updating product data on the blockchain, multiple factors come into play, including the base transaction cost, execution opcodes, and data storage costs. For example, storing comprehensive product data directly on Ethereum significantly escalates the gas required due to the multiple "SSTORE" operations for each field in the product struct. In contrast, leveraging IPFS for data storage and only saving the product's Content Identifier (CID) on the blockchain can drastically reduce gas costs, making it a more economical choice.

Chapter 3. Bibliographic Research

3.1. Blockchain

Blockchain is the latest and most promising technology in the modern economy, offering one of the safest ways to handle data. It is essentially a decentralized database, or distributed ledger, that precisely logs all digital events or transactions that take place and are independently confirmed among members of a network. The immutability and irrevocability of transactions throughout a business ecosystem are guaranteed by this technology. Its innovative approach to safe and effective value exchange is highlighted by its dependence on a network consensus model for transaction validation, where trust among transaction participants is based on cryptographic techniques.

In the book "Mastering Bitcoin" [7], M. Antonopoulos describes blockchain as a sequential chain of validated blocks, each one linked to the previous block, all the way back to the initial or "genesis" block. A block is a data structure that contains a header, which stores metadata. This metadata typically includes the address of the participant who added the block, a reference to the preceding block, and a unique digital signature, although the specific details may vary depending on the blockchain implementation. The data contained within each block represents a list of all transactions that have been approved for inclusion in that particular block.

A key element of blockchain technology are consensus algorithms, which operate as a way for network users to concur on the legitimacy of transactions without the need for a central authority. These algorithms guarantee that every distributed ledger copy is maintained in sync with every network node. Proof of Work (PoW) and Proof of Stake (PoS) are two of the most often utilized consensus techniques. Proof of Work, which is used by Bitcoin, requires miners to solve complex cryptographic puzzles, consuming significant computational resources. In addition to adding a new block to the blockchain and receiving payment in bitcoin, the first miner to solve the problem wins. In contrast, Proof of Stake chooses validators based on how many coins they own and are willing to "stake" or lock up as collateral. This method is seen as more energy-efficient compared to Proof of Work.

Public key cryptography, also known as asymmetric cryptography, is a security feature to uniquely identify participants in the blockchain network. This mechanism generates two sets of keys for network members. The public key is openly available and is used to encrypt messages or verify digital signatures. In contrast, the private key is kept secret and is used to decrypt messages or digitally sign documents. When someone wants to securely send a message to the intended recipient, they encrypt the message using the receiver's public key. Only the person who holds the matching private key - in this case the receiver - can then decrypt the encrypted message and see the original content. This method ensures confidentiality and integrity, since only the intended recipient can access the message. Additionally, digital signatures made with the private key can be verified by

anyone who has the public key, providing authentication and non-repudiation. Through this system, cryptography enables secure communication and data exchange over insecure channels.

Even though blockchain has strong security capabilities, it can still be vulnerable to attacks in some situations. The 51% attack is the most widely known form of assault on a blockchain system. It entails assembling a majority coalition of participants who possess the network's processing capacity, enabling them to exploit the network for their own ends. It typically targets implementations of the Proof of Work (PoW) protocol because, when nodes get contradictory data from two adjacent nodes in the system, one corrupt and one legitimate, they will accept the neighbor that can validate the most blocks. If 51% of the network's computing power verifies a block, it will most rapidly advance to the next one, because of the majority control.

3.1.1. Bitcoin

Bitcoin, the world's first decentralized digital currency, was unveiled to the public in 2009 by a mysterious individual or group going by the pseudonym Satoshi Nakamoto. Despite numerous investigations and conjecture over the years, the genuine character behind this alias continues to be one of the biggest enigmas of the digital era.

Over the past 10 years, it has grown significantly, gained global traction and come under careful examination. Bitcoin has drawn the interest of the world's financial community and sparked the creation of a large number of additional cryptocurrencies and blockchain-based inventions.

The total amount of bitcoins is 21 million and they can be created only through the process of mining, using the Proof of Work (PoW) consensus algorithm. Every bitcoin transaction requires transferring value between digital wallets. Each transaction uses digital signatures matching the sender's address to secure it. This guarantees that only bitcoin owners can spend their coins. Bitcoin's innovative fusion of cryptography, decentralization, and agreement among network users makes it a safe and trustless peer-to-peer payment system. This removes the need for middlemen such as banks.

3.1.2. Ethereum

As the authors state in the article [8], Ethereum, unveiled in late 2013 by Vitalik Buterin, represents a second-generation blockchain that expands beyond the capabilities of its predecessor, Bitcoin. Ethereum introduces a platform for developing decentralized applications (DApps) and smart contracts, which are self-executing contracts with the terms of the agreement directly written into code.

This innovation allows for a wide range of applications beyond mere financial transactions, including decentralized finance (DeFi) platforms, non-fungible tokens (NFTs), and more, thereby creating a more versatile blockchain ecosystem. Ethereum's native cryptocurrency, Ether (ETH), is used to compensate participating nodes for computations performed, as well as for transaction fees and services on the network.

Unlike Bitcoin, which focuses on being a decentralized digital currency, Ethereum's broader goal is to become a global platform for decentralized applications, offering developers the tools to build and deploy smart contracts and DApps with ease. This groundbreaking approach has positioned Ethereum at the forefront of blockchain development, fueling its growth and adoption across various sectors.

Smart contracts are an important innovation in blockchain technology that was first thought of in the 1990s. The goal was to digitize and enforce agreements without needing centralized legal entities. Smart contracts are programmed to automatically carry out the terms of a contract on a blockchain. This allows secure, direct transactions between parties who don't necessarily trust each other. It does this by removing the need for intermediaries and the costs involved with them.

These digital contracts have major advantages over traditional methods. They lower risks in transactions and administrative expenses. They also improve business efficiency. By using blockchain's transparency and security, smart contracts simplify operations. They offer a more dependable and efficient way to fulfill contractual obligations.

3.2. Data Storage

Storage and time problems are considered issues for a decentralized storage systems such as Ethereum. Therefore, decentralized storage systems should be designed well to guarantee these aspects for network members. Although blockchains, as decentralized technologies, help to ensure data immutability, they face challenges related to storage capacity and transaction processing time [9].

A widespread problem in large, decentralized networks is managing data storage, particularly as the need for storage space in the system increases. This issue frequently comes up in many networks. Public blockchains are often viewed as a solution for maintaining unchangeable data within such decentralized systems. However, blockchains have their own constraints, most notably the lack of memory to store huge amounts of data in a decentralized way. This restriction is seen as a major worry within blockchain technology.

Blockchain technology involves many processes that take a lot of time. Every transaction needs to be checked through a peer-to-peer system. As the number of blocks grows, this can cause big delays. The steps in this process like confirming transactions, running the consensus algorithm, adding a new block, putting the transaction info into that block, sending copies to all users, and then waiting for other users to accept it, all make blockchain technology slow.

The HyperText Transfer Protocol (HTTP) is a fundamental technology of the World Wide Web that enables communication and data transfer between web browsers and servers. It works through a centralized system where data is stored and accessed from specific locations or URLs. While this centralized method is effective for the traditional web, it presents major difficulties when used with blockchain technology, which has a decentralized nature and functions through a peer-to-peer network.

The InterPlanetary File System (IPFS) is a decentralized file sharing protocol introduced in 2015 by Juan Benet. It works through a network of computers running IPFS software, enabling anyone to participate by either operating an IPFS node or by using the network to store and access files. IPFS is flexible, supporting many file types like documents, audio, video, and images. Unlike the standard HTTP system where files are located by their web address, IPFS finds files based on their content using a unique hash generated for each file when uploaded. This means identical files will have the same hash, allowing efficient file retrieval and verification by comparing the locally computed hash with the one on IPFS, ensuring content integrity.

In IPFS, nodes have the capacity to pin content to ensure its continuous storage and accessibility. Nodes can also remove content that hasn't been used recently to save

storage space. As such, each node in the network only keeps the content it's interested in, along with indexing information to identify which node has specific content.

Adding a new version of a file to IPFS generates a unique cryptographic hash and a new Content Identifier (CID). This structure guarantees files in IPFS are protected from tampering and censorship. Modifications to a file make a new version without replacing the original.

Since IPFS is decentralized, content availability depends on at least one node actively hosting it. Public gateways, which aren't dedicated hosting services, may remove content with low demand. Therefore, to ensure continuous content availability, individuals must run their own IPFS node.

According to a study [10], using IPFS for storing data in Blockchain based applications performs better by optimizing the memory and reducing the transaction delay. The costs are also minimized, being 10% less than in the case of storing the data on-chain.

3.3. Alternative Anti-Counterfeit Technologies

3.3.1. QR Codes

QR codes have been used as a simple way to provide product information and confirmation of authenticity. However, their usefulness for combating counterfeit goods has some clear weaknesses.

Since QR codes are easy to copy, counterfeiters can take the code from a real product and put it on a fake, misleading buyers. This limits how well QR codes work on their own as an anti-counterfeiting tool. Even though they are convenient for accessing info and checking authenticity by scanning, QR codes alone don't stop counterfeiting without extra protections like encryption or links to secure verification databases.

3.3.2. Radio-Frequency Identification tags

There have been many tries for finding a good solution to minimize the production and selling of counterfeit products. Radio-Frequency Identification (RFID) tags are one of them. RFID tags are electronic devices that use radio frequency waves to transmit data stored on a microchip, enabling wireless identification and tracking of objects.

These microchips enable the verification of an item's authenticity, thus acting as a strong defense against counterfeiting. The paper [11] explores the idea of making RFID tags unclonable by inextricably linking them to a Physical Unclonable Function (PUF). This innovation leads to the development of security protocols necessary for verifying a product's genuineness when equipped with such a system, concentrating on offline authentication due to its practical benefits.

Despite being useful, RFID technology has some drawbacks. A major worry is privacy. RFID tags can be read at a distance without consent, creating a risk of people or items being tracked without approval. In addition, RFID systems have high initial setup and upkeep costs, making it a big investment for companies. The technology also faces technical difficulties, like signal interference from metals or fluids, which can hinder tag readability and dependability. Furthermore, the environmental impact of disposing RFID tags, many containing metals and other non-biodegradable substances, raises sustainability issues.

3.3.3. AI-Based Technologies

Another solution would be systems that use machine learning, specifically image and text recognition, to provide consumers with an affordable and easy way to verify products without needing special equipment. [12]

Users simply photograph product packaging containing text, logos, and certification marks, and submit the images. The images are processed and verified by a backend system composed of a web server and machine learning application. The machine learning component needs pre-trained deep learning models to recognize valid logos and marks. This enables the precise identification of counterfeit marks, even those with small changes, significantly improving the detection of fake products. By allowing consumers to easily verify products with just a smartphone, this solution is an effective tool in the fight against product piracy.

This approach also faces challenges. These include the large amount of data required to train the models and the long process of gathering and manually labeling that data. Also, the detection time of 3.1 seconds per mark may be too slow for real-time verification, which is a practical constraint. There is also a balance between speed and precision, as faster algorithms can reduce reliability. The effectiveness of this method heavily relies on user participation and the quality of images given for review.

Chapter 4. Analysis and Theoretical Foundation

Supply chain administration faces problems like redundant services, poor coordination between departments, and an absence of consistent benchmarks because of inadequate transparency. The issue of fake goods is increasingly widespread, making it nearly impossible to identify counterfeits by visual inspection alone. These fakes present considerable obstacles for legitimate businesses, yet many don't fully understand the impact counterfeits have on brands.

The method presented in this thesis aims to improve the detection of counterfeits by monitoring their supply chain history, using Blockchain technology. This method ensures the verification and traceability of genuine products across the supply chain. A Blockchain system decentralizes data, enabling multiple parties to access it at the same time. A key benefit is the difficulty of altering recorded information without agreement from all involved parties, thereby safeguarding data security and protecting against vulnerabilities.

4.1. Ethereum Blockchain

Ethereum represents a major improvement in blockchain technology by expanding its capabilities beyond just tracking cryptocurrency transactions like Bitcoin. It introduces a flexible platform that works as a distributed state machine, capable of storing and managing diverse types of data in a general-purpose data store. This ability allows Ethereum to hold any data that can be expressed as key-value pairs.

However, Ethereum is unique because it can execute code within its state machine, tracking every state change on its blockchain. This innovation enables complex contracts called smart contracts and encourages the development of decentralized applications (dApps).

4.1.1. Keys and Addresses

Ethereum has two main account types: Externally Owned Accounts (EOAs) and contract accounts. EOAs assert ownership of ether using digital private keys, Ethereum addresses, and digital signatures. The Ethereum account addresses are generated from the private keys. Specifically, each private key corresponds to exactly one Ethereum address, which is also referred to as an account.

The security of Ethereum is based on a pair of cryptographic keys: a private key and public key. The private key is a randomly generated sequence of numbers that is kept confidential by the user. It acts as a proof of ownership and is used to authorize transactions. On the other hand, the public key is mathematically derived from the private key and can be shared with other users. This pairing allows for secure digital interactions, permitting users to demonstrate ownership of their Ethereum accounts without exposing sensitive details.

An Ethereum address is a shortened version of the public key that identifies user accounts on the Ethereum blockchain. These addresses are created by hashing the public key cryptographically, which generates a 42-character hexadecimal string starting with "0x". Ethereum addresses have multiple uses: receiving transactions, interacting with smart contracts, and indicating the destination for sending Ether and tokens. In contrast to the keys they come from, addresses do not have information about the owner's identity, offering some privacy in transactions.

The uniqueness of the private key is ensured by using a method to randomly generate 256 bits. Since the process of deriving the public key from the private key uses a highly irreversible function, it is impossible to obtain the private key if you only know the public key. The only way to compute the private key from the public key is through brute force, by trying every possible value for the private key until you find the one that generates the public key. However, this approach is not feasible due to the enormous 256-bit address space.

The function utilized to generate the public key from the private key is highly irreversible. It takes the private key as input and generates a point on an elliptic curve, allowing easy calculation of the public key from the private key. However, the private key cannot be calculated from the public key. The same principle applies to the address generation function. The address is created by applying the Keccak-256 cryptographic hash function, which is a variant of SHA-3. Like the elliptic curve function, this hash function is also one-way. Therefore, the public key cannot be deduced if only the address generated from it is known.

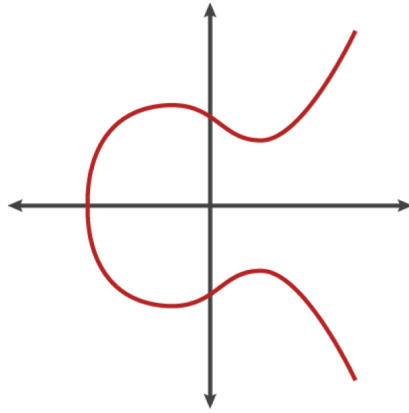


Figure 4.1: A visualization of an elliptic curve

The function K calculates the point on the elliptic curve. K can be written as:

$$K(k) = k \cdot G, \quad (4.1)$$

where k is the private key for the account, and G is a fixed point called the generator point. The multiplication between the private key and G is not regular multiplication, but rather elliptic curve multiplication. Elliptic curve multiplication is straightforward to apply in one direction, but infeasible to reverse when trying to find a specific value K^{-1} .

Another one-way function is Keccak-256, which is used on the public key to create the address. Unlike elliptic curve multiplication, Keccak-256 is a cryptographic hash function.



Figure 4.2: Address creation using Keccak-256 hash function

4.1.2. Transactions

Transactions are important parts of the Ethereum network that help change its state and run contracts on the Ethereum Virtual Machine. They are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain [13].

Transactions contain key information like nonce, gas price, gas limit, receiving address, value, data payload, and parts of an ECDSA digital signature. These pieces, carefully serialized using the Recursive Length Prefix (RLP) encoding method, make up the core of transaction communication within the Ethereum network.

The nonce is a crucial part of Ethereum transactions. It functions as a sequence number generated by the external Ethereum account sending the transaction. By increasing with every transaction, the nonce guarantees each transaction is unique. This protects against unauthorized or duplicate transactions which could harm the integrity of the network.

The gas price is the monetary reward paid by the person who initiates the transaction to the network for processing the transaction. It is measured in wei, which is the smallest unit of ether, and indicates how much the initiator is willing to pay per unit of gas used during the execution of the transaction.

The gas limit indicates the maximum amount of gas that the person who started the transaction is willing to use for running the transaction. It acts as a protection against uncontrolled computation or unexpected actions within smart contracts. Choosing a suitable gas limit makes sure transactions have enough computing power to complete while avoiding too much use that could cause out-of-gas errors and failed transactions.

The recipient is the Ethereum address that the transaction is sent to. It identifies the destination, which could be an externally owned account (EOA) or a smart contract address, that will obtain and execute the transaction's instructions. Transactions can be directed to either EOAs, which are accounts owned and controlled by individuals, or smart contract addresses, allowing for value transfers, interactions with contracts, and decentralized app (dApp) functions within Ethereum.

The value field represents the quantity of ether (ETH) sent from the person initiating the transaction to the receiving address. It indicates the financial worth linked to the transaction and enables direct transfer of value, asset trade, and contract settlements within Ethereum's network.

The data field allows for the inclusion of variable-length binary data with the transaction. This enables the transaction to carry extra information, parameters, or instructions needed for contract interactions, decentralized app (dApp) execution, and custom transaction functionalities. The data payload can include many data types, such

as function calls, transaction metadata, and arbitrary message content. This expands the flexibility and usefulness of Ethereum transactions by letting them transmit more than just value transfers between accounts.

The *v*, *r*, and *s* elements make up the ECDSA digital signature of the external Ethereum account sending the transaction. They function as cryptographic evidence that the transaction creator is who they claim to be and has approved the transaction. The *v* part indicates the recovery ID. The *r* and *s* parts are signature pieces calculated from the private key belonging to the person sending the transaction. These pieces guarantee the transaction is authentic and unchanged.

4.1.3. Consensus

Consensus is a fundamental concept in blockchain technology that maintains agreement and consistency amongst network participants with respect to the state of the distributed ledger. It enables multiple nodes in a decentralized network to reach a common understanding of the valid transactions and the order in which they are added to the blockchain [14].

There are many consensus algorithms that are used, with each one having its own distinct qualities, advantages, and tradeoffs. Choosing the right consensus algorithm depends on factors such as security requirements, scalability, energy usage, decentralization, and the specific use case of the blockchain network. Some of the most well-known consensus algorithms in the blockchain space include: Proof of Work, Proof of Authority and Proof of Stake.

Proof of Work

The Proof of Work (PoW) consensus algorithm involves a prover (requestor) and a verifier (provider). The prover performs computationally demanding tasks to accomplish a goal. After completing the tasks, the prover presents the solution to the verifier or multiple verifiers. This process allows the network to reach consensus through an asymmetric workload, where the prover does the heavy computational lifting and the verifiers simply check the result.

In PoW, miners compete to solve tricky mathematical problems through mining. This requires massive computing power and energy use. Miners endlessly generate different cryptographic hash values until they find one that satisfies the criteria. Once a valid solution is discovered, it gets added to the blockchain, safeguarding the network by making tampering with previous blocks difficult.

However, PoW has downsides. Its computational calculations lead to high energy consumption, raising environmental issues. Moreover, the reliance on computing power can centralize mining power, with a few large operations controlling the network. This problem worsens with the development of specialized hardware, known as Application-Specific Integrated Circuits (ASICs), which further excludes smaller participants and compromises decentralization and security.

Proof of Stake

Proof of Stake (PoS) is another method for reaching consensus in blockchain networks. Unlike PoW, which depends on computing power and energy-intensive mining,

PoS works by validators putting up their cryptocurrency holdings as collateral to validate transactions and create new blocks.

In PoS, validators are selected to generate new blocks based on the number of coins they have and are willing to stake or lock up. The chance of being picked as a validator is proportional to the amount staked. This is meant to motivate validators to act properly, since they have a financial interest in the network's integrity. Validators get transaction fees and newly created coins as rewards for helping with block creation.

One big advantage of PoS over PoW is its energy efficiency. Since PoS doesn't require miners to do computationally intensive work, it uses much less energy, making it more sustainable for blockchain networks. Also, PoS aims to address centralization issues by enabling more participants to become validators, rather than concentrating power with a few large mining entities.

However, PoS also has challenges and limits. One concern is the "nothing-at-stake" problem, where validators have little to lose by supporting multiple competing blockchain forks, which could hurt network security. To reduce this risk, PoS systems often have penalties for validators who act badly or try to support conflicting chains.

The proof of stake algorithm proposed for Ethereum is called Casper.

Proof of Authority

In a Proof of Authority (PoA) network, validators are chosen based on their reputation and standing within the network. These validators tend to be well-known entities or organizations that have been authorized to take part in validating blocks. Their role is to verify transactions and add them to the blockchain.

A key characteristic of PoA is its emphasis on identity verification and trust. Validators in a PoA network must go through a strict verification process to demonstrate that they are reliable and able to maintain the integrity of the network. This verification often requires providing proof of identity, reputation, and commitment to upholding the network's protocols and standards.

PoA networks are frequently utilized in private or consortium blockchains where trust between participants is already established, such as within companies or between business partners. By relying on trusted validators, PoA networks can achieve high transaction throughput and low latency, making them suitable for use cases that prioritize efficiency and scalability over decentralization.

4.1.4. Wallets

As noted by Andreas M. Antonopoulos [13], a wallet is a software application that functions as the primary user interface to Ethereum. The wallet regulates access to a user's funds, managing keys and addresses, monitoring the balance, and generating and signing transactions.

In the Ethereum system, private keys are essential for protecting wallets. These secret codes are utilized to authorize transactions, verifying ownership of the connected assets. If a private key becomes known by others, it can result in unauthorized transactions and possibly major asset loss. As a result, keeping private keys confidential and intact is vital for any Ethereum user to prevent theft or loss of funds. Ethereum wallets can be categorized into two primary types: deterministic and non-deterministic.

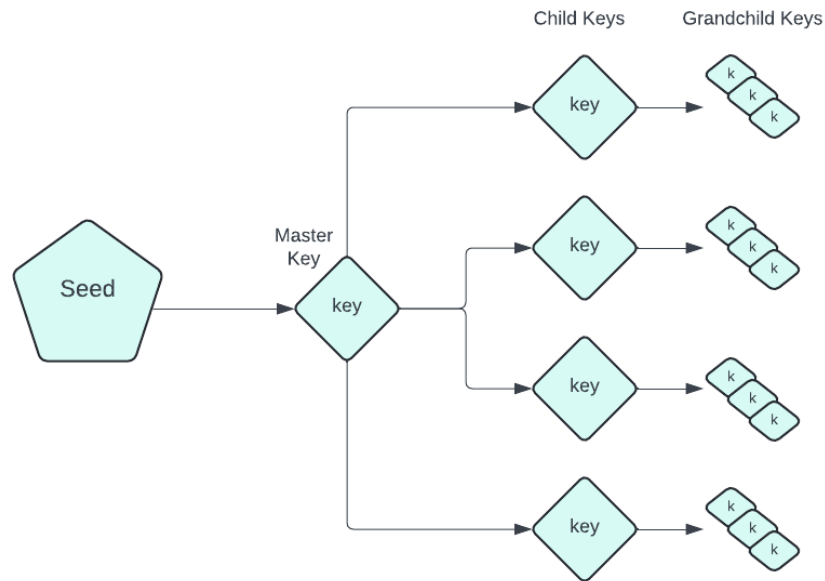


Figure 4.3: Hierarchical key generation

Deterministic wallets derive all their keys from a single master key, called the seed. This approach offers an easy way to back up and restore wallets if needed. Deterministic wallets regularly use mnemonic phrases to recover access. This gives a handy way to get a wallet back if data is lost. The mnemonic method is both easy to use and safe, as long as the phrase stays private and protected. Mnemonic phrases are sensitive data. If someone gets the phrase, they can recreate the wallet and control its assets.

Non-deterministic wallets create keys separately from one another, without a shared starting point. This method, also called "Just a Bunch of Keys" (JBOK), does not permit regenerating keys from one origin. Although this approach can give extra privacy since each key is unconnected, it needs more work to back up every key one by one.

4.1.5. Smart Contracts

A smart contract is a computer program having self-verifying, self-executing, tamper-resistant properties [15]. A key feature of smart contracts is that once they are deployed on the blockchain, their code cannot be changed. This ensures trust and predictability, as users can be sure the contract's logic will not unexpectedly alter. Despite this, the values and state variables stored within the contract can change as a result of transactions and interactions with the contract.

Smart contracts are also deterministic, meaning that with the same inputs and blockchain state, they will always produce the same result. This determinism is crucial in a decentralized system, where thousands of nodes run identical code to maintain consensus across the network. Each node operates its own instance of the Ethereum Virtual Machine (EVM), which provides the limited context for smart contracts to run. The EVM allows smart contracts to access their own internal state, data about the transaction that started their execution, and limited information on recent blockchain blocks.

Smart contracts are written using a particular programming language, with Solidity being the most widely used. These smart contracts can only execute on the Ethereum Virtual Machine after being compiled and converted into bytecode format. The compila-

tion process also produces an Application Binary Interface (ABI), which is a JSON file describing the contract's functions to enable interaction with the contract through function calls from transactions or other smart contracts. The bytecode is the value found in the data field of the transaction used to deploy the contract to the blockchain.

Solidity provides a range of features designed to support the creation of smart contracts. Functions are the building blocks of smart contracts, allowing developers to define the contract's behavior and operations. Modifiers are used to control function access and enforce specific rules, enhancing contract security. Solidity has certain keywords that define how a function will behave based on how it is declared. Functions can be declared using the following keywords:

- **View:** A function with this keyword does not modify the contract's state, indicating it can only read data without altering the blockchain.
- **Pure:** Functions that are declared as pure do not access or change the contract's state. Pure functions only perform calculations using the parameters passed to them.
- **Payable:** When a function is declared payable, it can accept Ether or other cryptocurrency as part of the transaction.

In addition to functions and modifiers, Solidity includes a mechanism for emitting events, allowing contracts to communicate with external systems. Events are logged on the blockchain and can be used to signal specific actions or states, facilitating interaction between smart contracts and off-chain applications.

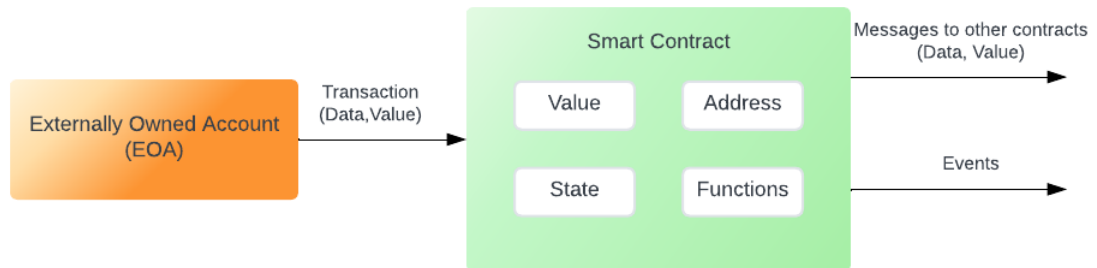


Figure 4.4: A basic structure of Smart Contract

4.2. InterPlanetary File System

The InterPlanetary File System (IPFS) is a protocol that does not rely on centralized servers. Instead, it uses a peer-to-peer network architecture to store and share files across many nodes. This decentralized approach makes the web more robust and efficient. With traditional web protocols like HTTP, files exist on individual servers. If those servers go down, the files become inaccessible. With IPFS, files are distributed across the network, having multiple copies in different nodes. This reduces the risk of failure and disruption.

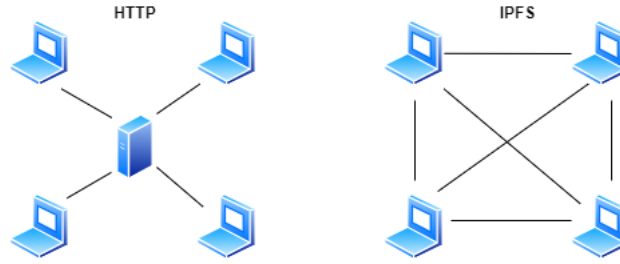


Figure 4.5: HTTP vs. IPFS

When information is uploaded to IPFS, it gets split into smaller chunks, usually around 256 KB in size. Each chunk is hashed using a cryptographic hashing algorithm like SHA-256. This hash, known as a content identifier (CID), uniquely labels the data. Even a small change in the content will result in a completely different hash, ensuring each CID is exclusive to the specific data it represents. This system solves the problem of duplicate files. Files that have identical content will be assigned the same address, so copies cannot exist on the same network computer.

These content identifiers become the basis for locating and retrieving data in IPFS. Instead of depending on a centralized server or URL, a CID points directly to the data itself, no matter where it's stored in the peer-to-peer network. This approach enables IPFS to be highly decentralized and resistant to censorship or single points of failure, as the same data can be stored across multiple nodes.

When a file is uploaded to IPFS, its content identifier is published to the distributed hash table (DHT). This keeps the information about which IPFS nodes are storing the file data. As new nodes join or existing nodes leave the IPFS network, the mappings in the DHT are dynamically updated to maintain consistent associations between CIDs and file locations. This ensures that files in IPFS remain discoverable and retrievable even as the overall network changes.

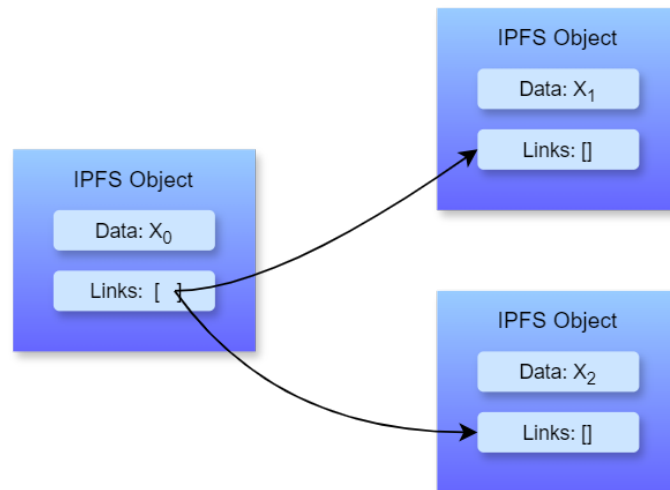


Figure 4.6: IPFS Objects

Objects in IPFS are fundamental building blocks that represent pieces of data and their relationships within the system. These objects form a Directed Acyclic Graph (DAG), allowing for complex data structures and flexible organization. An IPFS object typically contains two key components: data and links.

The data field of an IPFS object can contain a small amount of content, like a file segment, metadata, or other important information. It is typically restricted to the size of 256KB, promoting a modular way to storing data.

The links field connects and reassembles parts of a file that have been distributed over a network. It holds a list of the connections to the different pieces of a file that is spread out across a network. When a file gets split up into chunks, the data structure used to represent the file's content becomes a Directed Acyclic Graph (DAG). A DAG has the characteristics of a Merkle tree. In a Merkle tree, the leaf nodes are labeled with a cryptographic hash of a data block, and the internal nodes are labeled with a hash of their child nodes.

4.3. QR Codes

A QR code (Quick Response code) is a type of matrix bar code or two-dimensional code designed to be read by smartphones. The code consists of black modules arranged in a square pattern on a white background. The information encoded may be text, a URL or other data [16].

Each QR code is designed with a specific structure comprising function patterns and an encoding region, all surrounded by a quiet zone border on all four sides.

The arrangement of specific elements within a QR code, known as function patterns, are essential for the scanning and decoding process. These patterns enable QR code readers to properly recognize and orient the code, even if it is rotated or located in different positions. There are four main categories of function patterns:

- **Finder Patterns:** A model for detection the location of the QR Code. By ordering this design at the three corners of a symbol, position, size, and angle of the symbol can be observed. This type of pattern consist a structure which can be Observed in all directions (360°) [17].
- **Separators:** The white spaces that surround the finder patterns and set them apart from the data area of the QR code. They create a distinct border between the finder patterns and the encoded data region. This clear separation improves the scannability and readability of the QR code by visually isolating the finder patterns.
- **Timing Patterns:** A model to find the center point of each cell in a QR code that has alternating black and white patterns. It can be used to fix the center coordinates of data cells when the symbol is distorted or there is an error in the spacing between cells. Corrections are made in both the vertical and horizontal directions.
- **Alignment Patterns:** Assist in correcting distortion and ensuring accurate decoding. These patterns are constructed with 5×5 dark modules, 3×3 light modules, and a single dark module at the center.

The encoding region is where the actual data is stored. It contains format information, version information, data, and error correction codewords. The format information is located near the top-left, top-right, and bottom-left finder patterns. The version information is stored in a 6×3 block above the bottom-left finder pattern and a 3×6 block to the left of the top-right finder pattern.

The quiet zone is a 4-module-wide area surrounding the entire QR code. This area must remain empty, containing no data or markings, to ensure that external text or graphics do not interfere with the scanning and interpretation of the QR code. The

quiet zone acts as a buffer, providing a clear boundary around the QR code to facilitate accurate detection and decoding.

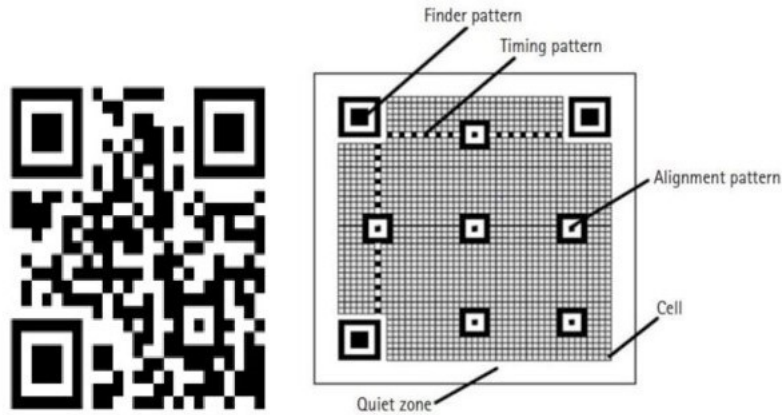


Figure 4.7: QR Code Structure [18]

QR codes offer a flexible and efficient method to encode and share data across many uses. Their design permits quick scanning, allowing users to access various types of information, from web links and contact details to payment data and media content.

The robust nature of QR codes, with inbuilt error correction, means they stay readable even under challenging conditions or when partly damaged. This durability, combined with the capacity to store significant amounts of information in a compact format, makes QR codes ideal for supply chains and various other applications.

Furthermore, QR codes are cost-effective to generate and eco-friendly, as they can decrease the need for printed materials. Their customizable essence permits branding and personalization, additionally extending their usefulness.

4.4. Use Cases and Scenarios

In this section, the main success scenarios and use cases will be described. Figure 4.8 illustrates how customers detect counterfeit products during the purchasing process. Customers can track a product's journey from its manufacturing point to an intermediary, where source and destination information are recorded on the blockchain. The product is intended to reach a legitimate retail destination. However, if a fraudulent actor duplicates the QR code and associates it with a counterfeit item, the retail destination will differ from the original source where the customer bought the product. This process ensures that customers can verify the legitimacy of their purchases through blockchain-based provenance tracking.

4.4.1. Use Case 1: Manufacturer adds a product

The sequence diagram from 4.9 illustrates the process of adding a product to the blockchain. The interaction begins with the manufacturer initiating the *addProduct()* action through the web application. The application then communicates with IPFS by

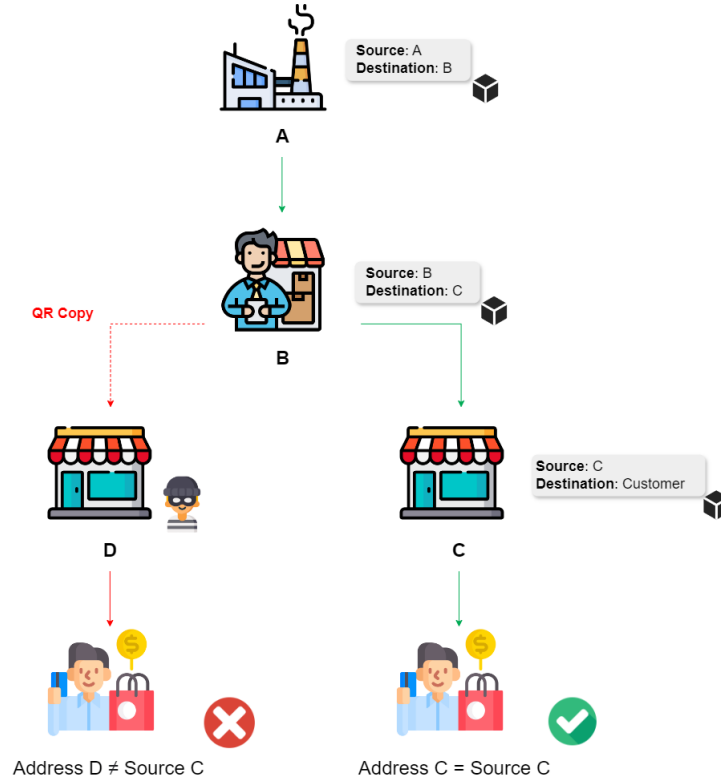


Figure 4.8: Handling Counterfeit Products

invoking the *pinToIps()* function, which stores the product's data in a decentralized manner.

Once the data is successfully pinned to IPFS, the system returns a Content Identifier to the web application. The web application subsequently uses this CID to call the *addProduct(CID)* function on the blockchain. This step records the product's CID on the blockchain, ensuring that the product's data is immutable and verifiable.

Following the successful addition of the product data to the blockchain, a confirmation message is sent back from the blockchain to the web application. Finally, the web application displays this confirmation to the manufacturer, completing the process.

4.4.2. Use Case 2: Supplier updates a product

The sequence diagram from [4.10](#) demonstrates the process of updating a product's information. The supplier initiates the *updateProduct(serialN)* action through the web application, where *serialN* represents the product's serial number.

The web application then interacts with the blockchain to retrieve the current product data by invoking *getProduct(serialN)*, which returns the old CID associated with the product. Using this old CID, the web application calls *getProductDetails(oldCID)* to fetch the existing product details from IPFS.

After obtaining the current product details, the supplier updates the product information, and the web application invokes the *pinToIps(updatedProd)* function to store this updated data in IPFS. This process generates a new CID for the updated product data.

Having the new CID, the web application calls *updateProduct(newCID)* on the

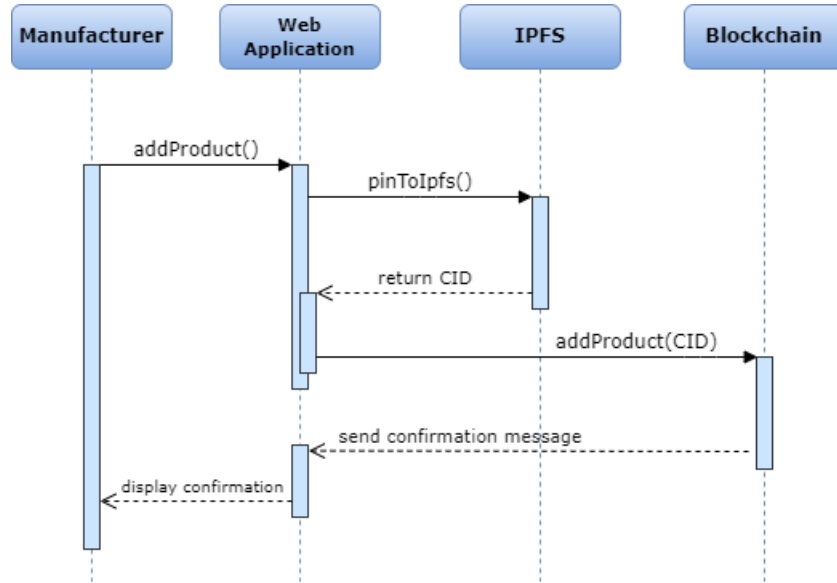


Figure 4.9: The process of adding a product

blockchain, which records the new CID, thereby updating the product’s information on the blockchain.

Upon successful updating of the product information, a confirmation message is sent from the blockchain back to the web application. In the end, the web application displays this confirmation to the supplier, indicating that the product update process is complete.

4.4.3. Use Case 3: Customer searches for a product

The sequence diagram [4.11](#) illustrates the process of retrieving the history of a product’s information. The supplier initiates the *searchProduct(serialN)* action through the web application, where *serialN* represents the product’s serial number.

The web application first queries the blockchain for events related to the addition of the product. The blockchain returns these events, which include the initial CID associated with the product. Using this initial CID, the web application calls *getProductDetails(initialCID)* to fetch the initial product details from IPFS.

Next, the web application queries the blockchain for events related to updates of the product. The blockchain returns these events, which include any updated CIDs associated with the product. Using these updated CIDs, the web application calls *getUpdatedProductsDetails()* to fetch the updated product details from IPFS.

Once all relevant product details are retrieved from IPFS, the web application consolidates this information and displays the product’s history to the supplier.

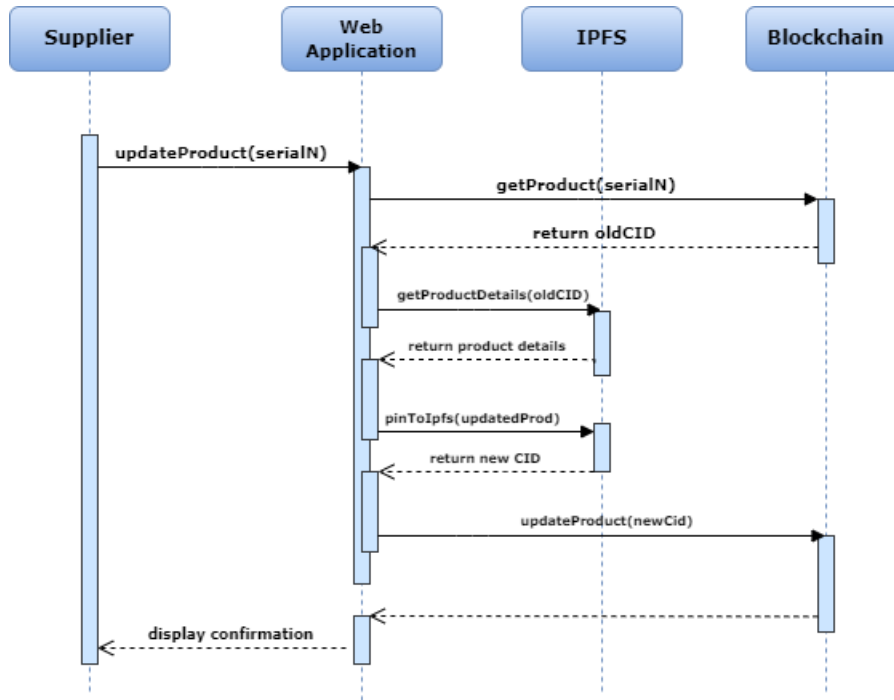


Figure 4.10: The process of updating a product

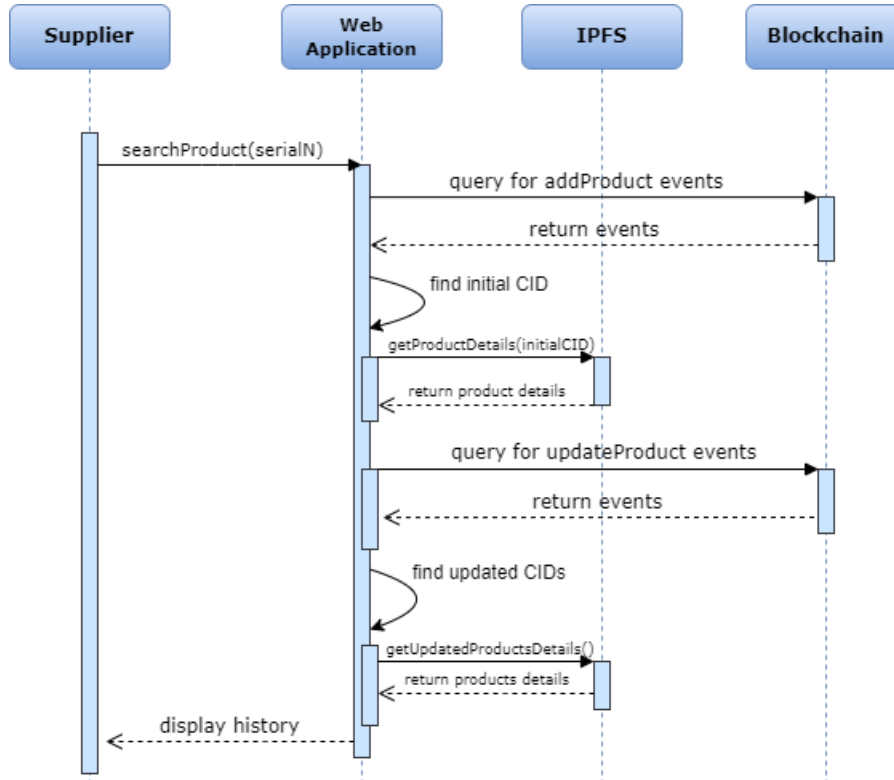


Figure 4.11: The process of searching for a product

Chapter 5. Detailed Design and Implementation

5.1. System Design

The system uses the blockchain technology to build a safe, decentralized network for tracking product details, decreasing counterfeiting, and improving transparency across supply chains. By recording product information on a blockchain, manufacturers and providers have an incorruptible, steady, and private way to uphold data integrity. Customers profit from this system by being able to access the full supply chain record of items, enabling them to confirm authenticity and guarantee they are not buying counterfeit products.

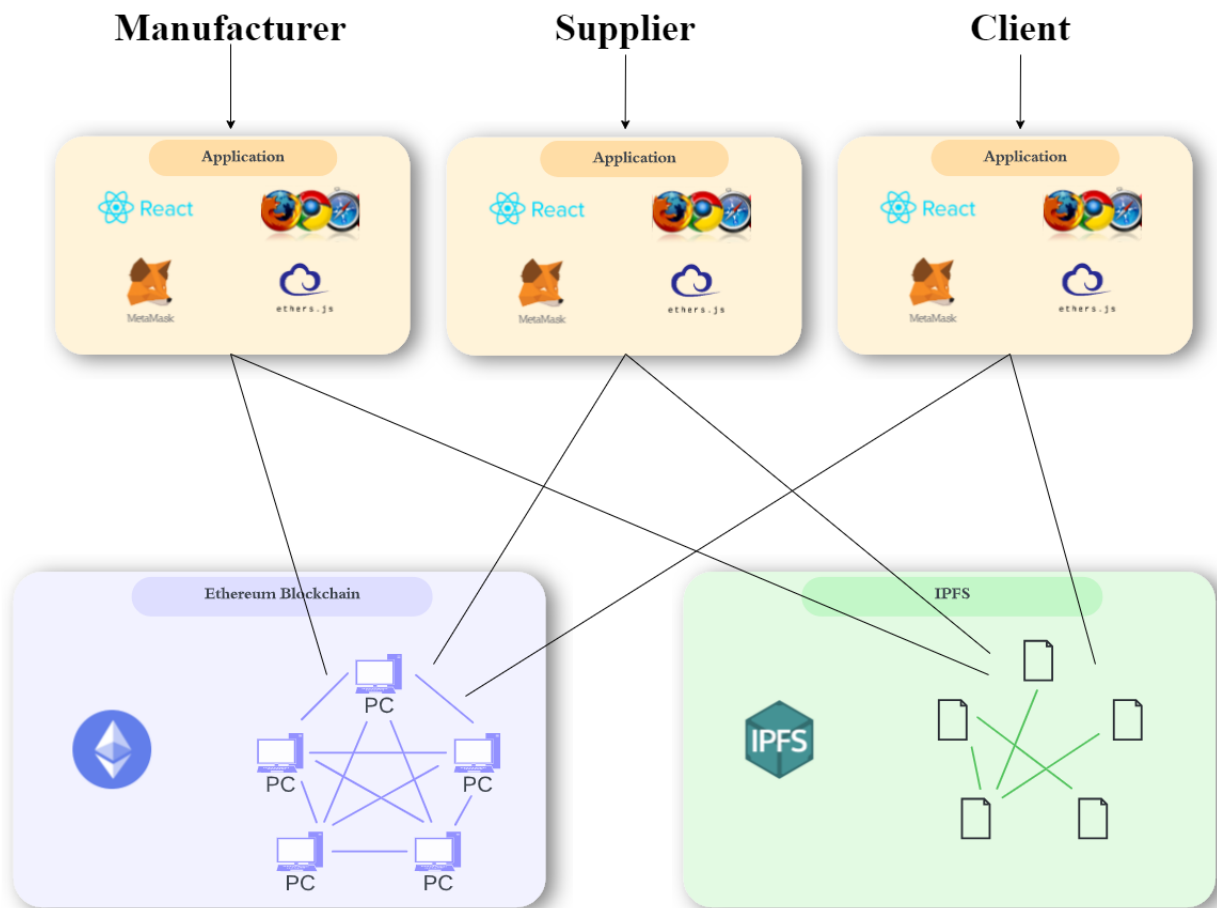


Figure 5.1: System architecture

5.1.1. Web Application

The application was built using React, a widely used JavaScript library known for its ability to create responsive user interfaces efficiently and adaptably. A major advantage of using React is its component-based design, which facilitates the construction of self-contained, reusable components that handle their own state. One major benefit of utilizing React in blockchain applications is its effective updating system. React's virtual DOM selectively updates only the sections of the web page that require modification, preventing unnecessary re-renders. This leads to smoother and quicker user experiences, which is vital in the application discussed in this thesis that needs to display real-time blockchain information, such as the states of transactions.

Websites constructed with React are also suitable with mobile browsers, thus making the sites functional on phones. This is also important for the application, since it strives to be operable on mobile devices too, to provide more adaptability to the end-user, who necessitates being capable to access the platform without a desktop computer. For instance, the customer should be able to scan a QR code directly in the store to instantly determine whether the product is authentic or counterfeit.

The implementation integrates Redux [19], which is a robust state management library that helps manage complex state in JavaScript applications. It promotes a predictable state management approach using a central store that holds the entire app's state. This centralized structure makes the app easier to maintain and debug since all state changes happen in one place and follow a clear order. Actions dispatched from any part of the application trigger pure functions called reducers, which take the current state and an action as input and return the new state. By separating the state from the UI, Redux makes it easier to extend, maintain, and monitor complex workflows involved in identifying and verifying counterfeit products.

The Ethereum-based JavaScript library Ethers.js [20] is vital for interacting with the Ethereum blockchain as it offers a wide set of capabilities required to send transactions, connect with smart contracts, and manage wallets. For example, it simplifies the creation of Ethereum providers and signers, enabling seamless interactions with the blockchain network. In the application, Ethers.js is used to connect to both Metamask and a JSON-RPC-based Ethereum node through Infura, ensuring robust and flexible connectivity choices. In addition, it assists in managing digital identities and performs essential operations like retrieving network statuses and user accounts, which are critical for verifying the authenticity of products in the supply chain.

MetaMask is a cryptocurrency wallet software that facilitates interaction with various blockchains [21]. It offers users the convenience of accessing their wallets via a browser extension or mobile app, enabling smooth interactions with decentralized applications (dApps). MetaMask supports multiple key management options, such as hardware wallets, to boost security by isolating the keys from online sites. For developers, MetaMask provides a globally accessible Ethereum API embedded into web3-compatible browsers. This API is crucial when users initiate transactions or require digital signatures, prompting a detailed MetaMask interface to ensure users are well-informed and to mitigate the risk of widespread security breaches. MetaMask introduces a global API into visited websites through *window.ethereum*, which allows these sites to request users' Ethereum accounts, read blockchain data, and prompt users to sign messages and transactions. Therefore, *window.ethereum* serves as a vital link between users' wallets and dApps. MetaMask comes pre-configured with quick access to the Ethereum blockchain

and various test networks through Infura, providing an immediate start-up without the need to synchronize a full node and offering enhanced security options for those who choose to connect their preferred blockchain provider.

Infura is a platform that offers software tools and application programming interfaces to help developers build decentralized apps and Web3 services. It acts as a Web3 backend provider, speeding up development by supplying ready-made infrastructure instead of requiring developers to build everything from scratch [22]. It uses a distributed network of cloud-hosted nodes to deliver enterprise-grade infrastructure. Through its cloud node network, Infura can interact with the blockchain for the developer. Its microservice architecture scales dynamically based on demand and lets developers interact with Ethereum via WebSockets or HTTPS. In this thesis project, the application uses Infura directly to ensure consistent functionality across different user scenarios. In the case when a user accesses the website without MetaMask or via a mobile device, the application maintains the ability to read data from deployed smart contracts through the Infura API endpoint. This setup guarantees uninterrupted access and interaction with the blockchain, regardless of the user's setup.

5.1.2. Ethereum Blockchain

The application offers users the flexibility to choose between various blockchain environments for executing transactions, specifically the Sepolia and Goerli testnets, as well as a localhost blockchain deployed with Hardhat.

The proposed system can run on the Sepolia testnet, which Ethereum core developers advise using for smart contract application development. Sepolia was initially launched in October 2021 as a proof-of-authority network, but later transitioned to a proof-of-stake consensus mechanism. [23] This change aimed to closely replicate the operational conditions of the Ethereum mainnet. With its permissioned validator set and as a relatively new test network with a small state, Sepolia provides fast sync times and requires less storage for running nodes versus the mainnet or older testnets. These features make it very suitable for quick interactions with the network and running nodes without extensive resource allocation.

In addition, the system is compatible with the Goerli testnet, another key test environment for Ethereum based applications. Launched in January 2019 by the Ethereum research team, Goerli serves as a proof-of-concept for the Ethereum 2.0 upgrade, enabling a reliable and efficient platform for testing new features and improvements. Goerli uses a Proof-of-Authority consensus and is known for its stability and resistance to forks, providing a resilient testing ground. [24]

These test networks are purposely created to mimic the environment of the main Ethereum network. This allows developers to deploy and evaluate the performance of smart contracts and decentralized apps without the need for extensive hardware. By making use of both Sepolia and Goerli testnets, the system utilizes existing blockchain frameworks to guarantee that all interactions and transactions occur in test settings that reflect real-world functionality. This optimizes development and testing productivity.

The application also enables deployment on a local blockchain setting, made possible by Hardhat, a versatile Ethereum development platform [25]. This configuration mimics a complete blockchain node on a local computer, furnishing an effective and efficient way for rigorous testing and building. Hardhat offers advanced features such as stack traces, console logging, and interactive debugging. The use of a local blockchain

reduces dependency on public testnets.

5.1.3. IPFS

The InterPlanetary File System (IPFS) significantly enhances the decentralization and efficiency of data storage. It functions as a peer-to-peer network designed for storing and exchanging data within a distributed file system. In contrast to conventional file storage systems that depend on centralized servers, IPFS operates by establishing a resilient network of node operators, each responsible for storing a portion of the overall data. This collaborative effort contributes to the development of robust, decentralized storage solutions.

Every file and data fragment stored on IPFS is distinguished by a unique content identifier, known as a CID, which is derived from the cryptographic hash of the file's contents. This approach guarantees that each piece of content can be retrieved using its hash, ensuring the integrity of the content from storage to retrieval.

In the context of this thesis project, IPFS is employed to store comprehensive information about each product within the supply chain. Upon uploading a product's details to IPFS, a CID is generated and securely stored in a smart contract on the Ethereum blockchain. By storing the CID on the blockchain instead of the actual data, the application's efficiency and scalability are significantly improved as data storage is delegated to IPFS. The blockchain serves as a verifiable ledger for these CIDs, enabling secure and transparent tracing of information such as product origins, processing stages, and shipment details.

5.2. System Implementation

5.2.1. Smart Contract

The smart contract for this thesis project is developed in Solidity, which is a high-level language specifically created for implementing smart contracts on the Ethereum blockchain. It offers static typing, support for inheritance, libraries, and the ability to define complex user-defined types. These features make Solidity an excellent choice for creating contracts for various purposes, including supply chain management.

Within this project, the smart contract named ***ProductTracker*** represents the final version, aiming to efficiently manage product data within the blockchain. To achieve this, the contract uses mappings to establish connections between product serial numbers and their respective content identifiers (CIDs). Additionally, it enables the tracking of complex products that consist of multiple components. Key functions are implemented to facilitate the addition and updating of both simple and complex products. These functions incorporate appropriate checks to ensure data integrity and prevent duplications.

Initially, an earlier version of the contract attempted to include all product fields directly within the blockchain. However, this approach proved to be inefficient due to the associated costs and slower transaction times caused by the larger amounts of on-chain data. As a result, the design was refined to store only the essential identifiers in the blockchain, while using IPFS for detailed data storage. This strategic shift significantly enhances efficiency and scalability. The evolution from the initial approach to the current model, as well as the differences between them, will be thoroughly discussed in later sections of the thesis.

The complete version of the smart contract is available for review in Appendix [A](#). Following is an extract from the contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ProductTracker {
    mapping(uint => string) products;
    mapping(uint => uint[]) complexProducts;

    // events

    function addProduct(uint _serialNumber, string memory _cid) public {
        require(!productExists(_serialNumber), "Product already exists");
        products[_serialNumber] = _cid;
        emit ProductTracker__AddProduct(_serialNumber, _cid, block.timestamp);
    }

    function updateProduct(uint _serialNumber, string memory _cid) public {
        require(productExists(_serialNumber), "Product not found");
        products[_serialNumber] = _cid;
        emit ProductTracker__UpdateProduct(_serialNumber, _cid, block.timestamp);
    }

    function addComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
        require(!productExists(_serialNumber), "Product already exists");
        for (uint i = 0; i < _containingProducts.length; i++) {
            require(productExists(_containingProducts[i]), "Component product not found");
        }
        complexProducts[_serialNumber] = _containingProducts;
        emit ProductTracker__AddComplexProduct(
            _serialNumber,
            _containingProducts,
            block.timestamp);
    }

    function updateComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
        require(productExists(_serialNumber), "Complex product not found");
        for (uint i = 0; i < _containingProducts.length; i++) {
            require(productExists(_containingProducts[i]), "Component product not found");
        }
        complexProducts[_serialNumber] = _containingProducts;
        emit ProductTracker__UpdateComplexProduct(
            _serialNumber,
            _containingProducts,
            block.timestamp);
    }

    // getters
}
```

Mappings in Solidity are collections that pair keys to values. Conceptually, a mapping is virtually initialized such that every possible key exists from the start. Each key is paired with a value whose byte-representation is all zeros, which corresponds to the default value of the value's type [\[26\]](#).

The *products* mapping in the *ProductTracker* smart contract is crucial for associating each product having a unique serial number, and its corresponding Content

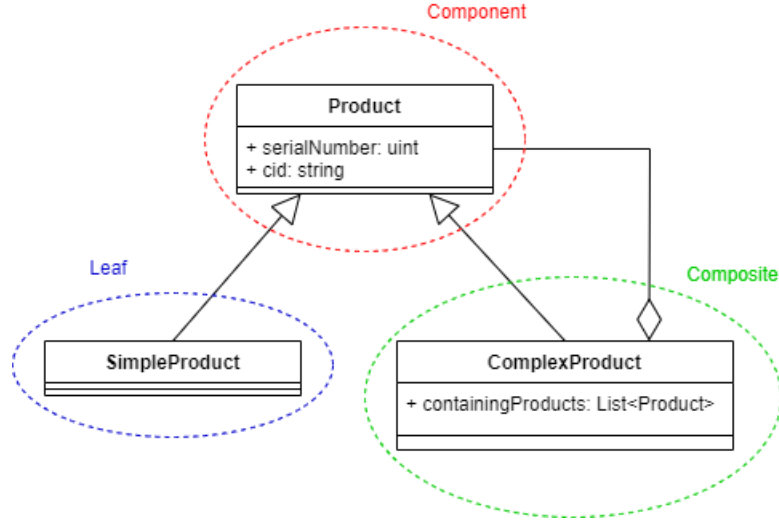


Figure 5.2: Managing products

Identifier (CID). The CID, which is a 46 characters long string, refers to detailed product information stored on IPFS. This configuration guarantees the security and accessibility of all product information by utilizing blockchain technology, thereby improving transparency and traceability in supply chain operations.

The **complexProducts** mapping expands the capabilities of the smart contract to handle intricate products composed of multiple other products. Each entry in this mapping connects a complex product’s serial number to an array of serial numbers, each representing one of its component products. This system employs a structure similar to the Composite Design Pattern, as illustrated in Figure 5.2, which is particularly well-suited for scenarios where individual objects and compositions of objects need to be treated in a uniform manner. The inclusion of complex products is a vital feature for enhancing the user experience. Some merchants attempt to deceive customers by selling multiple products in bundles, making it challenging to verify the authenticity of the items within. In such cases, some products might be genuine while others could be counterfeit, leaving customers unaware unless they have access to advanced tracking and verification systems. By using the proposed system, even if a product bundle contains multiple items, each can be individually verified as genuine, ensuring that customers receive exactly what they are paying for, thus safeguarding their health, safety, and economic interests.

In Solidity, events provide a way to log and notify external entities, such as user interfaces or other smart contracts, about specific occurrences within a smart contract. They serve as a mechanism for recording data onto the blockchain, making it transparent and easily accessible. Emits, on the other hand, are used to trigger or emit these events within the smart contract code. Events are similar to logs or records that capture important information, enabling external observers to react to them and obtain relevant data. These events are stored on the blockchain as part of the transaction history, and can be retrieved even after the transaction has been completed [27].

The **ProductTracker_AddProduct** event is triggered when a new product is added to the **products** mapping. It logs the product’s serial number, its CID, and the timestamp of the transaction. The system retrieves data from the blockchain by querying the emission of these specific events, ensuring the information remains synchronized.

When the information of a current product is changed in the smart contract,

the ***ProductTracker_UpdateProduct*** event is triggered. This allows for tracking and reacting to modifications in product information, guaranteeing that the system can retrieve the latest details about any product. The logged variables are the product's serial number, the CID, and the timestamp of the transaction.

The two other events are similar but they aim to track the lifecycle of complex products by recording their creation and any later updates. The information that is logged represents the serial number of the complex product, a list of serial numbers of all the included products and the timestamp of the transaction.

The ***addProduct()*** function is designed to register a new product within the blockchain. It takes two parameters: *serialNumber*, which is a unique identifier for the product, having exactly eight characters, and *cid*, which is the content identifier, pointing to the IPFS location where the product's detailed data is stored. The function begins by checking if a product with the given serial number already exists in the contract's state, using the ***productExists*** function. If the product does exist, the function execution is halted, and an error message "*Product already exists*" is returned to the user. If the product does not exist, the function maps the product's serial number to its CID in the products mapping. This operation effectively registers the product. Upon successful registration, the function emits the ***ProductTracker_AddProduct*** event.

The ***updateProduct()*** function enables the updating of an existing product's details on the blockchain. It takes two parameters: *serialNumber*, which is the unique identifier for the product, and *cid*, the updated content identifier where the new product details are stored. The function first checks if the product's serial number exists by calling the ***productExists*** function. If the product is not found, the function halts and returns the error message "*Product not found*". If the product is found, the function updates the products mapping with the new CID for the given serial number. In the end, the function emits the ***ProductTracker_UpdateProduct*** event.

The ***addComplexProduct()*** function registers a new complex product that is made up of multiple existing products. It takes two parameters: *serialNumber* and *containingProducts*, which is an array of serial numbers representing each component that makes up the complex product. The function first checks if the complex product is already registered in the blockchain's registry by using the ***productExists*** function. If the product is already registered, the function returns the error message "*Product already exists*". Next, the function goes through each serial number in the *containingProducts* array to verify that all components are already registered in the blockchain. If a component is not found, the function stops and returns the error message "*Component product not found*". In the case when all components exist, the function registers the complex product by mapping its serial number to the array of component serial numbers in the *complexProducts* mapping and emits the ***ProductTracker_AddComplexProduct*** event.

The ***updateComplexProduct()*** function modifies the details of an existing complex product. This function takes two parameters: *serialNumber* and *containingProducts*, an array of serial numbers for the new set of components that will make up the complex product. The function first checks that the complex product specified by the *serialNumber* actually exists in the blockchain registry. If the product is not found, the operation is halted, and the error message "*Complex product not found*" is returned. After that, the function validates each serial number in the *containingProducts* array to ensure that every component is already registered in the blockchain. If a component is not found during this check, the function returns "*Component product not found*". When all components are confirmed to exist, the function updates the mapping by assigning the new array of

component serial numbers to the complex product's serial number. The last step is to emit the ***ProductTracker_UpdateComplexProduct*** event.

The contract also includes several utility functions designed to facilitate access and verification of product data stored on the blockchain. The ***getProduct()*** function retrieves the cid associated with a given product serial number, so that details about products can be accessed on IPFS. The ***getComplexProduct()*** function has a similar role for complex products, returning an array of serial numbers that represent the individual components of the complex product. The ***productExists()*** function is used to verify the existence of a product in the blockchain. It checks if there is a non-empty entry corresponding to a provided serial number, returning a boolean value that indicates whether the product is registered in the system.

5.2.2. Smart Contract Deployment

Deployment is the process of publishing a compiled smart contract to a blockchain. This involves generating a special transaction that includes the contract's bytecode. The transaction is then signed and sent to the blockchain, where it is executed by the network's nodes. Once the execution is successful, the contract is stored on the blockchain at a specific address. This address is later used to interact with the deployed contract. For deploying the *ProductTracker* smart contract, two methods were tested.

The first method consists in deploying the smart contract using Hardhat, a tool designed to facilitate Ethereum development tasks like compiling, deploying, and testing smart contracts. The following script was implemented:

```
const { ethers } = require("hardhat");

async function main() {
  console.log("Deploying the smart contract...");
  const ProductTracker = await ethers.getContractFactory("ProductTracker");
  const accounts = await ethers.getSigners();

  // Deploy the contract
  const productTracker = await ProductTracker.connect(accounts[0]).deploy();
  await productTracker.deployed();

  console.log(`ProductTracker is deployed at address: ${productTracker.address}`);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

The script can be run using the commands:

```
> npx hardhat node
> npx hardhat run deploy.js --network localhost
```

The smart contract can also be deployed on other networks, but this needs to be specified in the Hardhat configuration file (usually `hardhat.config.js`) so it includes

the network settings. This involves specifying the network details, including an Remote Procedure Call URL to connect to the network and the private keys of the accounts needed for deploying the contracts.

The second method for deploying the smart contract involves using Remix IDE [28], an open-source web application that provides an accessible environment for developing Ethereum smart contracts within a browser. Remix IDE is a good choice because it is easy to use. There is no need to set up or install anything, allowing developers to write, compile, and deploy smart contracts using a simple and intuitive interface.

Remix integrates well with Metamask, enabling users to deploy contracts to any Ethereum network, including testnets and the mainnet, directly from the browser. The IDE includes features like static analysis, debugging tools and a suite of plugins.

5.2.3. Interactions with the Blockchain

The first action that the system does when is started is to load the blockchain data. This is accomplished through the following implemented functionalities:

```
export const loadProvider = (dispatch) => {
  const providerUrl = process.env.REACT_APP_INFURA_PROVIDER_URL;
  let connection;
  if (typeof window.ethereum !== "undefined") {
    connection = new ethers.providers.Web3Provider(window.ethereum);
  } else {
    connection = new ethers.providers.JsonRpcProvider(providerUrl);
  }
  dispatch({ type: "PROVIDER_LOADED", connection });
  return connection;
};
```

The function *loadProvider()* creates a connection to the blockchain, which is essential for the application to perform operations such as transactions or querying blockchain data.

The function first determines the appropriate method to connect to the Ethereum network. It checks if the user's browser has an Ethereum wallet integration, like MetaMask, which is a common setup for users actively interacting with blockchain applications. If such an integration is available, the function uses it to create a direct connection, enabling the application to use the user's existing wallet for blockchain interactions.

If no wallet is detected, a predefined connection URL from an environment variable is used. This URL points to Infura, which provides a remote access node to the Ethereum network. This setup allows the application to interact with the Ethereum blockchain without requiring the user to have a wallet extension installed, making it more accessible to a broader audience.

Once the connection method is established, the function updates the application's state to reflect that the blockchain provider is successfully loaded and ready for use. This state update is important for the rest of the application to know that blockchain functionalities are available and can be used for further operations. The connection object itself is then returned, providing a gateway for the application to conduct blockchain transactions and queries directly through the established connection.

```
export const loadNetwork = async (provider, dispatch) => {
  const { chainId } = await provider.getNetwork();
  dispatch({ type: "NETWORK_LOADED", chainId });
  return chainId;
};
```

The *loadNetwork()* function determines and manages the network to which the application is connected. It uses the provider sent as a parameter to obtain the chain ID of the network. This ID is important since it indicates the Ethereum network the application is running on, such as the mainnet or testnet.

The function updates the state with the most recent information after receiving the chain ID and sending this data to the application's state management system. The function returns the chain ID at the end so that other parts of the application can use it when needed.

```
export const loadAccount = async (provider, dispatch) => {
  const accounts = await window.ethereum.request({ method: "eth_requestAccounts" });
  const account = ethers.utils.getAddress(accounts[0]);
  dispatch({ type: "ACCOUNT_LOADED", account });
  return account;
};
```

The *loadAccount()* function starts by retrieving the account addresses directly from the user's Ethereum wallet, which is responsible for managing the user's accounts. Once the account addresses are obtained, the function takes the first account address. The function then updates the application's state by dispatching the verified account information, making it accessible throughout the entire application. The account address is returned. This functionality is needed such that the user knows exactly which account he is using.

```
export const loadProductTracker = (provider, address, dispatch) => {
  const product_tracker = new ethers.Contract(address, PRODUCT_TRACKER_ABI, provider);
  dispatch({ type: "PRODUCT_TRACKER_LOADED", product_tracker });
  return product_tracker;
};
```

The function *loadProductTracker()* function takes the provider and address as parameters and effectively establishes a connection to the smart contract. The first step is to initiate a new instance of the *ProductTracker* smart contract by using its application binary interface (ABI) and the blockchain provider that will facilitate the interactions. Through this setup step, the application and smart contract are effectively linked, enabling the application to run functions and monitor blockchain events. A dispatch call is performed to update the status of the application when the contract object has been instantiated.

The function finally returns the instance of the contract. The methods and events specified in the smart contract may be accessed directly from this returned object, which simplifies the application's ability to communicate with the blockchain.

```

export const submitProduct = async (
  serialNumber, name, sourceAddress, destinationAddress,
  remarks, provider, product_tracker, dispatch) => {
  let transaction;
  dispatch({ type: "NEW_PRODUCT_LOADED" });

  try {
    const signer = provider.getSigner();
    const signerAddress = await signer.getAddress();
    const block = await provider.getBlock("latest");

    const jsonProduct = {
      serialNumber: serialNumber, timestamp: block.timestamp, productName: name,
      sourceAddress: sourceAddress, destinationAddress: destinationAddress,
      remarks: remarks, manufacturer: signerAddress,
      supplier: "0x0000000000000000000000000000000000000000",
    };

    const cid = await pinJSONToIPFS(jsonProduct);

    transaction = await product_tracker
      .connect(signer)
      .addProduct(serialNumber, cid);
    await transaction.wait();
  } catch (error) {
    dispatch({ type: "NEW_PRODUCT_FAIL" });
  }
};

```

The *submitProduct()* function is the one that facilitates the addition of product details on the Ethereum network. It takes as parameters the details of a product that the user inserts, the provider, dispatch and an instance of the smart contract.

The first thing that the function does is to signal the initiation of a new product entry by dispatching a *NEW_PRODUCT_LOADED* action. After that, the signer's address is retrieved from the provider and the timestamp is extracted from the latest block. This timestamp is used to keep track of when the product was registered. All this data is assembled into a JSON object, which will be stored on the InterPlanetary File System to ensure integrity and availability. The address of the manufacturer will be the address of the person who signs the transaction and the supplier's address will be 0x00, since the product was not yet forwarded. The returned CID represents the location of the product data on IPFS. The final step is to create and execute a transaction through the instance of the smart contract connected with the signer's wallet and effectively add a new product to the blockchain.

If any error occurs during the execution, the function will signal this by dispatching a *NEW_PRODUCT_FAIL* action.

```

export const updateProduct = async (
  serialNumber, productName, sourceAddress, destinationAddress,
  remarks, provider, product_tracker, dispatch) => {
  let transaction;
  dispatch({ type: "UPDATE_PRODUCT_LOADED" });
  try {
    const signer = provider.getSigner();

```

```

const signerAddress = await signer.getAddress();
const block = await provider.getBlock("latest");

const oldCid = await product_tracker.getProduct(Number(serialNumber));
const oldProduct = await fetchJSONDataFromIPFS(oldCid);

const jsonProduct = {serialNumber: serialNumber, timestamp: block.timestamp,
  productName: productName, sourceAddress: sourceAddress,
  destinationAddress: destinationAddress, remarks: remarks,
  manufacturer: oldProduct.manufacturer, supplier: signerAddress};

const newCid = await pinJSONToIPFS(jsonProduct);
transaction = await product_tracker
  .connect(signer)
  .updateProduct(Number(serialNumber), newCid);
await transaction.wait();
} catch (error) {
  console.log("Error updating product", error);
  dispatch({ type: "UPDATE_PRODUCT_FAIL" });
}
};

```

The *updateProduct()* function updates the details of an existing product on the blockchain system. It takes as parameters the new details of a product that the user inserts, the provider, dispatch and an instance of the smart contract.

The function starts by dispatching the *UPDATE_PRODUCT_LOADED* action to the application's state management system, signaling that an update process has been initiated. After that, the signer's address is retrieved from the provider and the timestamp is extracted from the latest block. The function retrieves the current *cid* of the product using its *serialNumber*. With this *cid*, it accesses the current product information that is stored on IPFS. After this, the new details of the product are embedded into a JSON object and saved again at a new address on IPFS. The manufacturer address remains the same and the supplier address corresponds to the signer of the transaction. A transaction is created and executed through the instance of the smart contract connected with the signer's wallet and the updated product is registered on the blockchain.

In case of any errors during the update process, the function logs the error and dispatches an *UPDATE_PRODUCT_FAIL* action to inform the application of the unsuccessful attempt.

```

export const submitComplexProduct = async (
  serialNumber, name, sourceAddress, destinationAddress, remarks,
  containingProducts, provider, product_tracker, dispatch) => {
  let transaction, complexProductTransaction;
  dispatch({ type: "NEW_PRODUCT_LOADED" });
  try {
    const signer = provider.getSigner();
    const signerAddress = await signer.getAddress();
    const block = await provider.getBlock("latest");

    const containingProductsCids = await Promise.all(
      containingProducts.map(async (product) => {
        const cid = await product_tracker.getProduct(product.serialNumber);
        return cid;
      }));
    const containingProductsData = await Promise.all(

```



```

    containingProductsCids.map(async (cid) => {
      const data = await fetchJSONDataFromIPFS(cid);
      return data; }));

const jsonProduct = {
  serialNumber: serialNumber, timestamp: block.timestamp, productName: name,
  sourceAddress: sourceAddress, destinationAddress: destinationAddress,
  remarks: remarks, manufacturer: signerAddress, supplier: signerAddress,
  containingProducts: containingProductsData};

const cid = await pinJSONToIPFS(jsonProduct);

complexProductTransaction = await product_tracker
  .connect(signer)
  .addComplexProduct(
    serialNumber,
    containingProducts.map((product) => product.serialNumber));
await complexProductTransaction.wait();

transaction = await product_tracker.connect(signer).addProduct(serialNumber, cid);
await transaction.wait();
} catch (error) { console.log("Error submitting product", error);
dispatch({ type: "NEW_PRODUCT_FAIL" });}};

```

The *submitComplexProduct()* function handles the addition of complex products to the system, where each complex product contains multiple products that already exist on the blockchain. It takes as parameters the details of a product, the containing products, the provider, dispatch and an instance of the smart contract.

The function first signals the start of the product submission process by dispatching *NEW_PRODUCT_LOADED* action. This is followed by the retrieval of the signer's address from the provider and the extraction of the timestamp from the latest block. For each containing product, the function retrieves the content identifier from the blockchain that points to its detailed data stored on IPFS. Each CID is then used to fetch the corresponding product data from IPFS, which is saved in *containingProductsData*. A new JSON object for the complex product is created incorporating both the newly entered details (like serial number, name, addresses) and the fetched data of the containing products. In this case, the manufacturer's address and supplier's address are the same, considering that the supplier is the only one able to create bundles of products. This new product is pinned to IPFS, and a new CID is obtained. The final product is added to both mappings from the smart contract, being registered as a complex product and also as a normal product, so it can be included in other bundles.

Throughout the process, if any error occurs an action *NEW_PRODUCT_FAIL* is dispatched to update the application's state about the unsuccessful product submission.

```

export const loadAllProducts = async (provider, product_tracker, dispatch) => {
  const block = await provider.getBlockNumber();
  const addedProductsStream = await product_tracker.queryFilter(
    "ProductTracker__AddProduct", 0, block);
  const addedProducts = addedProductsStream.map((event) => event.args);

  const updatedProductsStream = await product_tracker.queryFilter(
    "ProductTracker__UpdateProduct", 0, block);
  const updatedProducts = updatedProductsStream.map((event) => event.args);

```



```

const products = addedProducts.filter((product) => {
  const updatedProduct = updatedProducts.find(
    (updatedProduct) =>
      updatedProduct.serialNumber.toString() === product.serialNumber.toString());
  return !updatedProduct;
});
updatedProducts.forEach((updatedProduct) => {
  products.push(updatedProduct);
});
dispatch({ type: "ALL_PRODUCTS", products });
};

```

The *loadAllProducts()* function has the role to retrieve all products that have been added or updated up to the latest blockchain state. This is needed in order to keep the application's state up to date. The function takes as parameters the provider, dispatch and an instance of the smart contract.

The first thing that the function does is to fetch the current block number from the blockchain using the provider. This guarantees that all transactions, up to the most recent block, are taken into account in the queries that follow. After that, it queries the blockchain for events of the type *ProductTracker__AddProduct*, which are emitted when a new product is added to the system. The query takes into consideration all events emitted from the genesis block up to the last one. The results are then processed to extract the arguments from each event, which contain the details of the added products. In a similar manner, the function retrieves events of the type *ProductTracker__UpdateProduct*, representing updates to existing products. This uses the same approach of querying and handling data as for newly added products.

For obtaining the final list of products, the function filters out any products from the added products list that also appear in the updated products list. The remaining products are the ones that are only added but not updated. This is done to prevent duplicate entries where an added product might also have been updated later. To this list are also added the updated products.

In the last step, the final list of products is dispatched to the application's state management system using the dispatch type *ALL_PRODUCTS*.

```

export const loadProductHistoryBySerialNumber = async (
  serialNumber, provider, product_tracker) => {
  const block = await provider.getBlockNumber();
  const addedProductsStream = await product_tracker.queryFilter(
    "ProductTracker__AddProduct", 0, block);

  const initialCid = addedProductsStream.find(
    (event) => event.args.serialNumber.toString() === serialNumber.toString());
  if (!initialCid) return { initialProduct: undefined, updatedProducts: [] };

  const initialProduct = await fetchJSONDataFromIPFS(initialCid.args.cid);

  const updatedProductsStream = await product_tracker.queryFilter(
    "ProductTracker__UpdateProduct", 0, block);

  const updatedCids = updatedProductsStream.filter(
    (event) => event.args.serialNumber.toString() === serialNumber.toString());
  const updatedProducts = await Promise.all(

```

```

    updatedCids.map((event) => fetchJSONDataFromIPFS(event.args.cid)));

    return { initialProduct, updatedProducts };
  };

```

The *loadProductHistoryBySerialNumber()* function retrieves the complete historical record of a product using its serial number. This function ensures that anyone interacting with the application can trace a product's origin and all updates that have been registered. It takes as parameter the serial number of the product, the provider and an instance of the smart contract.

The function fetches the current block number from the blockchain using the provider, ensuring that all transactions are taken into consideration. After that, it queries the blockchain for *ProductTracker__AddProduct* events, considering all blocks, from the genesis one up to the last. These events are filtered by the product's serial number that interests the customer.

If the function does not find an event for the initial product registration, it returns an object that indicates there is no initial product and an empty array of updated products. This will be used to inform the customer that the product was never registered in the system. Otherwise, it proceeds to fetch the detailed product data from IPFS using the content identifier stored in the event's arguments. This data represents the product details at the time of its addition.

Next, the function queries for *ProductTracker__UpdateProduct* events, which are emitted when a product's details are updated. It again filters these by the serial number to find all updates relevant to the specific product. The content identifier stored in the event's arguments is used to fetch the updated details from IPFS.

The function returns an object containing two properties: *initialProduct*, which holds the data of the product as first added, and *updatedProducts*, an array containing all later updates. This structure is suitable because a product can be added to the system only once but may be updated multiple times.

5.2.4. Interactions with IPFS

```

const pinJSONToIPFS = async (jsonData) => {
  const url = process.env.REACT_APP_PINATA_URL;
  try {
    const response = await axios.post(url, jsonData, {
      headers: {
        pinata_api_key: apiKey,
        pinata_secret_api_key: apiSecretKey,
        "Content-Type": "application/json"
      }
    });
    return response.data.IpfsHash;
  } catch (error) {
    console.error("Error pinning JSON to IPFS: ", error);
    throw error;
  }
};

```

The function *pinJSONToIPFS()* is used to upload JSON data to the InterPlanetary File System using the Pinata cloud service [29], which facilitates easier interaction with IPFS. It takes as parameter the JSON object that will be sent to IPFS.

The function performs an asynchronous HTTP POST request using the **axios** library. The function sends the JSON data to the specified Pinata URL and when the

request is successfully executed, it expects to receive a response that includes an IPFS hash. This hash is a unique identifier corresponding to the uploaded content on IPFS. The function returns this IPFS hash, which is used in the application to reference the stored data.

```
async function fetchJSONDataFromIPFS(cid) {
  const url = `https://gateway.pinata.cloud/ipfs/${cid}`;
  try {
    const response = await axios.get(url);
    return response.data;
  } catch (error) {
    console.error("Failed to fetch JSON data:", error);
  }
}
```

The *fetchJSONDataFromIPFS()* function is used to retrieve data from the Interplanetary File System using a specific content identifier. It takes as a parameter the cid of the wanted product.

The function constructs a URL using the base URL of the Pinata IPFS gateway concatenated with the provided cid. The cid is a unique identifier for data stored on IPFS, and accessing this URL allows the function to retrieve the data associated with that cid.

A HTTP GET request is made and if it is successful, the product details stored at that address is returned.

5.2.5. QR Code Generation

```
const handleGenerateQR = () => {
  const qrObject = {
    serialNumber: serialNumber,
    productName: productName,
    sourceAddress: sourceAddress,
    destinationAddress: destinationAddress,
    remarks: remarks,
  };
  const qrValue = JSON.stringify(qrObject);
  setQrValue(qrValue);
};
```

The *handleGenerateQR()* function wraps up the key product details into a QR code format, making it easier to share and verify this information. It starts by gathering product-related data such as the serial number, product name, source and destination addresses, and any additional remarks, and compiling this into a *qrObject*. This object is then converted into a JSON string. The resulting serialized string is then stored in the application's state. This state update allows for the generation and display of a QR code in the user interface, which can be scanned to quickly access and verify the encoded product information.

Chapter 6. Testing and Validation

This chapter centers on evaluating the solution in terms of cost, gas consumption, and system scalability. The analysis involves a comparative study of two distinct smart contract designs, providing insights into their performance and efficiency.

6.1. Experimental Setup

The application uses Sepolia test network, which ensures a robust and secure environment for testing smart contracts and transaction flows without the costs associated with mainnet deployments. The setup involves deploying smart contracts on this test network, which is responsible for recording and verifying each step in the product lifecycle. Sepolia uses the Proof-of-Stake consensus mechanism and the block time is approximately 12 seconds [30].

In this first scenario, all product data is directly stored within the smart contract on the Sepolia test network. This method guarantees that all information is securely and immutably recorded on the blockchain, providing transparency and reliability. However, storing large amounts of data on the blockchain can incur significant costs due to the high gas fees associated with Ethereum's data storage. Therefore, the evaluation of this scenario will focus on the gas consumption for various operations, including the addition, updating of product data and contract creation.

The second scenario integrates an additional connection to the InterPlanetary File System, facilitated through the Pinata API. In this configuration, product details are stored off-chain on IPFS, while only the IPFS hash of these files is stored on the blockchain. IPFS offers a decentralized storage solution that is both efficient and scalable, significantly reducing the on-chain storage requirements and associated costs. The Pinata API streamlines interactions with IPFS, making it simple to upload, pin, and manage files, thereby enhancing the overall efficiency of the system.

To evaluate costs, a comparison will be made between the expenses required to store product details directly on the blockchain and the costs associated with storing only the IPFS hash of these files. This analysis will consider the gas fees for both scenarios, aiming to identify the most cost-effective solution for data storage. The evaluation will include different types of operations, such as creating new records and updating existing ones, providing an extensive evaluation of the cost implications.

Scalability will be evaluated by analyzing key metrics such as gas consumption per operation and transaction throughput. This analysis will create an overview of the scalability of each solution, enabling the identification of the optimal approach for managing large volumes of product data within the application.

6.2. Experimental Results

The integration of systems for identifying counterfeit products onto the blockchain poses significant challenges, especially concerning cost and scalability. Such systems necessitate frequent blockchain updates and transactions to log and verify the authenticity of products. The integration's sustainability is challenged by the potential for high expenses due to gas fees and other associated costs. The evaluation concerning the cost implications of storing data on the blockchain compared to using IPFS will involve a detailed analysis of the two scenarios.

6.2.1. Storing files on the Blockchain

The implementation of the smart contract analyzed in this scenario can be checked in Appendix [B](#).

For estimating the gas cost of adding or updating a product, there are several factors that need to be considered, including the base cost of the transaction, the costs of executing the opcodes used in the function, and the costs related to storing data in the smart contract's state.

Computing the approximate gas cost for adding or updating a product involves understanding the key operations that consume gas and estimating their costs based on the Ethereum's Yellow Paper [\[6\]](#) and typical gas usage for similar operations. While exact numbers can vary based on the Ethereum network's state and the specific inputs to the function, we can provide a rough estimate.

For the case of storing all the data on the blockchain, the gas cost increases proportionally with the number of fields inside the contract. Every transaction has a base cost of 21,000 gas. Storing a new "Product" involves multiple "SSTORE" operations. Given that the "Product" struct has 8 fields, and assuming all fields are set from zero to non-zero values (which is the costliest scenario), each "SSTORE" operation costing 20,000 gas would result in:

$$8 \text{ fields} \times 20,000 \text{ gas} = 160,000 \text{ gas} \quad (6.1)$$

Emitting events also consumes gas. Assuming an average cost, this might add around 15,000 - 20,000 gas. The function execution includes simple arithmetic operations, memory allocation for the parameters, and mapping access. These costs are relatively low compared to storage costs but can add up. Estimating generously, we allocate 10,000 - 15,000 gas. Adding these up provides an approximate total of 216,000 - 226,000 gas, which is a large amount for a single transaction.

In addition, the cost increases for complex products. The higher the number of fields, the more "SSTORE" operations are required, each contributing to the overall gas cost. This makes direct on-chain storage increasingly expensive as the complexity and volume of product data grow.

On the other hand, for computing *Transactions Per Second*, we need to consider the maximum block size in gas, which is equal to 30,000,000 according to [\[31\]](#). For storing the details of a simple product, the gas used would be around 220,000. By using the formula

$$\text{numberOfTransactions} = \frac{\text{blockGas}}{\text{transactionGas}} \quad (6.2)$$

the resulting number of transactions for adding a product would be **approximately 136**. Given that the Ethereum's block mining rate is 12 seconds, the transaction throughput can be computed as follows:

$$\text{TPS} = \frac{\text{numberOfTransactions}}{\text{blockTime}} = \frac{136}{12} \approx 11 \quad (6.3)$$

The value obtained above denotes the number of transactions per second that can be accommodated within a single block.

6.2.2. Storing files using IPFS

The implementation of the smart contract analyzed in this scenario can be checked in [A](#).

This method implies storing only the Content Identifiers (CIDs) of the products in a mapping, and then emitting an event with the product details. The primary gas-consuming operations here are the storage operation (using the "SSTORE" opcode) and the event emission. The hash of a file published on IPFS has 46 characters and has the following structure:

QmTaFDZ8nMj79LXLorSGN7WtLmS7jvkud66gm.x5BXnfXcy

In Ethereum, the cost of storing a 256-bit word is 20,000 gas when changing from zero to a non-zero value. The CID, being a 46-character string, likely spans more than one 256-bit word, but for simplicity, we will estimate it as fitting within one word. Therefore, the basic storage cost would be 20,000 gas for storing the CID.

Additionally, emitting an event also consumes gas. The events include several parameters, but since event log data costs are relatively lower compared to storage, this might add a smaller amount to the total gas cost, depending on the size of the data being emitted. A rough estimate might place this in the range of a few hundred to a few thousand gas.

Taking into account these considerations, the primary components contributing to the gas consumption for product storage are the storage operation for the CID and the generation of the associated event. Storing a 46-character string as a CID and emitting the associated event could likely result in a gas cost in the vicinity of 21,000 (base transaction cost) + 20,000 ("SSTORE" for CID) + additional gas for the emitted event and operational overhead. This gives a result reaching around 50,000 to 60,000 gas. These approximations were also confirmed by performing multiple operations of adding and updating products using Remix IDE.

In terms of scalability, the number of transactions for adding a product that can fit into one block, according to the formula [6.2](#), is equal to:

$$\text{numberOfTransactions} = \frac{30,000,000}{55,000} \approx 545 \quad (6.4)$$

The number of transactions per second for this scenario is:

$$\text{TPS} = \frac{545}{12} \approx 45 \quad (6.5)$$

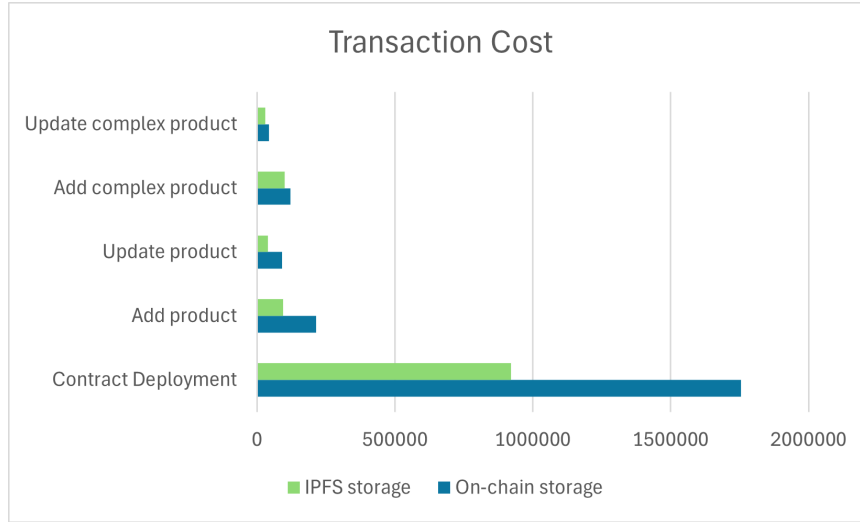


Figure 6.1: Transaction cost in gas

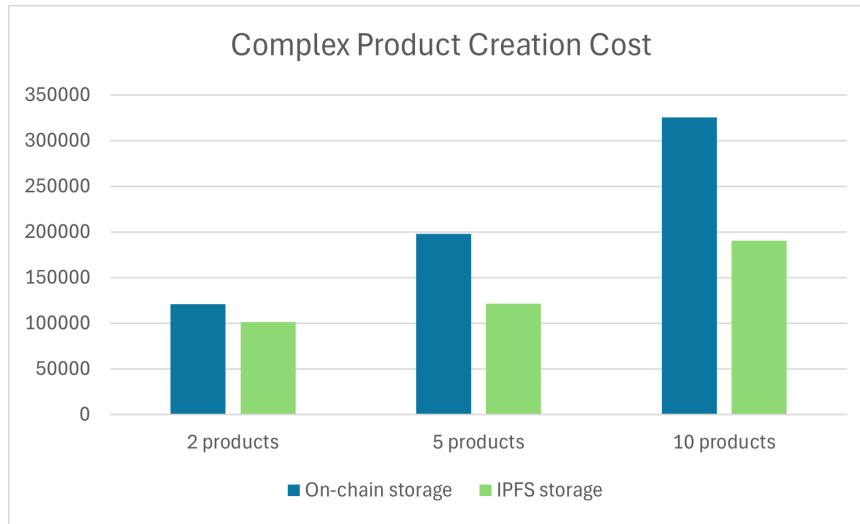


Figure 6.2: ComplexProduct Creation Cost in gas

6.2.3. Analysis

As presented in Figure 6.1, IPFS consistently demonstrates lower storage consumption than on-chain storage across various operations (contract deployment, adding and updating products, and managing complex products). IPFS reduces storage needs by a significant margin: approximately 47.5% for contract deployment, 56.4% for adding a product, 56.4% for updating a product, 16.4% for adding a complex product, and 31.6% for updating a complex product. These indicate that IPFS is a more resource-efficient option, which can lead to cost savings and improved scalability.

The analysis of gas costs for creating complex products using on-chain and IPFS storage reveals a consistent advantage for IPFS. From Figure 6.2 it can be seen that for 2 products, IPFS storage requires significantly less gas compared to on-chain storage. This efficiency gap widens as the number of products increases, with IPFS showing substantial gas savings for both 5 and 10 products. Given that the operation of adding a complex product is important in this application, it is crucial for this process to be as efficient as possible.

The TPS analysis provides a clear comparison of the scalability for both scenarios. When files are stored directly on the blockchain, the system can handle approximately 11 transactions per second. This relatively low TPS is due to the high gas consumption per transaction, which significantly limits the number of transactions that can be included in a single block.

On the other hand, using IPFS to store files and recording only the content identifiers on the blockchain results in a much higher TPS of approximately 45. This improvement is primarily because the gas cost per transaction is significantly lower when only storing CIDs instead of entire files.

These results highlight the trade-off between on-chain storage and off-chain storage with respect to scalability. Storing files directly on the blockchain incurs higher costs and lower throughput, while integrating IPFS for file storage and recording only CIDs on the blockchain provides a more scalable solution.

For applications that necessitate the storage of vast amounts of data, such as the one outlined in this thesis, the integration of IPFS and blockchain technology provides an optimal balance between cost-effectiveness, scalability, and data integrity. Given the significant difference in gas costs between storing data directly on the blockchain versus referencing off-chain data through content identifiers, the latter strategy represents a superior approach. This method not only reduces the costs associated with blockchain transactions but also enhances scalability by reducing the volume of data stored on-chain.

The increased transactions per second achieved through this approach ensures that the system can handle a greater number of transactions efficiently, maintaining responsiveness and performance even as data and transaction loads increase. Consequently, for applications requiring the storage of voluminous data, such as systems designed to combat counterfeit products, the utilization of IPFS in conjunction with blockchain technology is highly effective.

Scenario	Operation	Gas Consumption	TPS
On-chain storage	Contract Deployment	1,755,313	1.42
	Add product	215,310	11.61
	Update product	91,165	27.42
	Add complex product	121,047	20.65
	Update complex product	43,944	56.89
IPFS storage	Contract Deployment	921,408	2.71
	Add product	93,970	26.60
	Update product	39,794	62.82
	Add complex product	101,220	24.69
	Update complex product	30,048	83.20

Table 6.1: Comparison of Gas Costs and TPS for On-chain and IPFS Storage Scenarios

Chapter 7. User's Manual

7.1. Installation Prerequisites

The final system is a web-based application that can be accessed through any web browser, making it available to users regardless of their computer's operating system or whether they are using a desktop or mobile device. To view and monitor a product's history, users only need a web browser. However, in order to perform transactions to the smart contract, manufacturers and suppliers need to connect their wallets by using the Metamask extension for a desktop browser or the Metamask application for mobile devices.

For running the application on localhost, the computer used must have Node.js installed and a package manager for handling software packages and dependencies. Git, a version control system, also needs to be installed for cloning the project's repository.

7.2. Installation Guideline

The steps for starting the application are the following:

1. Add the Metamask extension to your browser and create a new wallet.
2. Send some SepoliaETH to your wallet by using Infura Faucet.
3. Use Git to clone the repository from [\[32\]](#).
4. Start a local Ethereum blockchain node by running `npx hardhat node`
5. Enter the **scripts** directory and run the command
`npx hardhat run deploy.js --network sepolia`
6. Copy the logged address and change it in **config.json** at the "11155111" key. If you want to use another network, repeat the previous step and change the address associated with the network's chain ID.
7. Run the command `npm install --legacy-peer-deps` inside the project's root directory.
8. Run the command `npm start` for starting the application.
9. The application is available at address `http://localhost:3000`

7.3. User Interaction

After completing the installation steps or accessing the application at the deployment address, the page presented in Figure [7.1](#) will be displayed.

The landing page offers the user three possibilities: proceeding as a manufacturer, supplier or a customer.

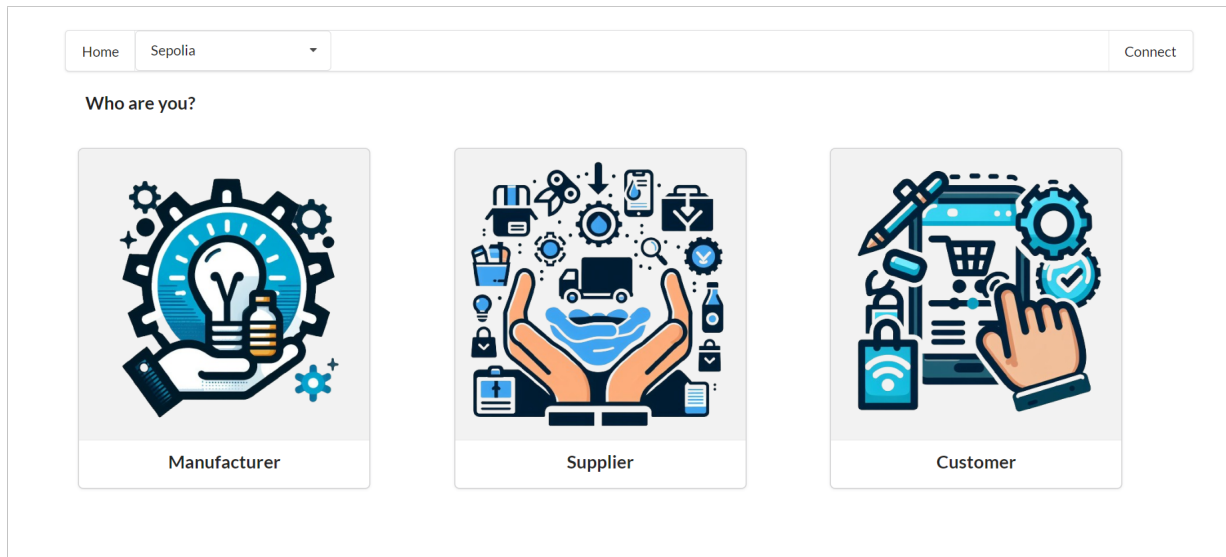


Figure 7.1: Landing Page

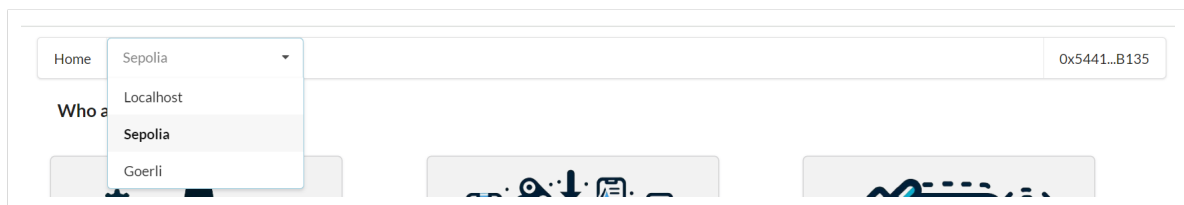


Figure 7.2: Menu bar

If the user is a manufacturer or supplier, he has to connect his wallet by using the Metamask extension. If the authentication succeeds, the menu bar will display the current user's wallet address.

A logged-in user can also switch between Ethereum networks if they want to make transactions on a network other than the one selected, as it can be seen in Figure 7.2. If another network is chosen, the Metamask extension will inform the user about the change.

The scenarios for all types of users will be presented:

7.3.1. Manufacturer

If the user is a manufacturer, he is able to add a new product to the blockchain, generate the QR code associated to this product, reject or confirm the transaction and see the transaction's details on Etherscan.

The page prompted is the one displayed in Figure 7.3. The manufacturer has to enter the product's details and after that, he can add the product to the blockchain. The QR code will be generated automatically after the product is added or the user can do it himself by pressing the button for this functionality.

The manufacturer will have the choice to confirm or reject the transaction after he sees all the fees needed for the operation, as presented in Figure 7.4. If he chooses to reject it, the use case ends here. Otherwise, after the transaction is confirmed by the network, the application will display a success message and the transaction address that can be checked on Etherscan, where more details are available.

[Home](#) Sepolia 0x5441...B135

Manufacturer Page

Serial Number

Product Name


Source Address

Destination Address

Remarks about the product

[Add Product](#)

[Generate QR Code](#)



[Download QR Code](#)

Figure 7.3: Manufacturer's page

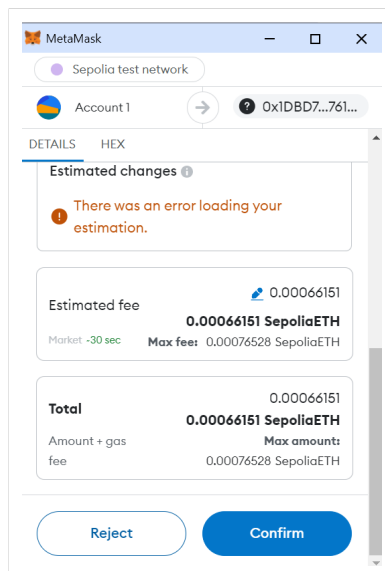


Figure 7.4: Metamask popup for confirming transaction

Figure 7.5: Supplier's page

Figure 7.6: Bundle page

The last step for the manufacturer is to download the QR Code. This code should be printed and stuck on the product that will be shipped to the destination warehouse.

7.3.2. Supplier

The supplier's page is presented in Figure 7.5. The supplier is an intermediate between the manufacturer and the final customer. His responsibility is to scan or upload the QR code of a product and add his details to the label, so the customer knows exactly the entire process his product went through.

If the supplier chooses to upload the QR code, a file explorer will pop up and he can select the desired image. The alternative is to use the integrated camera from his device. After the code is loaded into the application, the fields are filled automatically and he only has to change the ones he influences. The last step is to update the product's details and confirm the transaction. This process is the same as described in the section for the manufacturer.

The supplier can also create bundles of products. These are complex products that contain multiple simple products that already exist. In this case, he can upload multiple QR codes and the associated products will be displayed as presented in Figure 7.6. If he decides to remove a product or if it was added by mistake, there is a button for this

Figure 7.7: Customer's page

functionality. The added products have their names displayed. The created bundle is registered in the system as a normal product, this process being the same as described before.

7.3.3. Customer

The customer, as the ultimate recipient of the product, holds the utmost interest in verifying its source. This user does not need to authenticate for accessing the page where he can search for a product's history. This page can be seen in Figure 7.7. The customer can choose between searching for a product by its serial number, scanning the QR code with the camera integrated to his device or uploading a picture with the code. After the code is loaded into the application, the serial number field is completed automatically.

The route of a product is presented in chronological order, as shown in Figure 7.8. Actions performed by the manufacturer and those performed by the supplier are distinguished using different icons. The details of a product are displayed in the initial state and after an update is executed. The information is accompanied by a timestamp indicating when each action was performed. Users can track changes to the product's status and verify the sequence of events. This ensures transparency and allows for accurate monitoring of the product's journey.

For making the application more portable and easy to use, there was also a mobile layout considered. This can be seen in Figure 7.9. This was considered an important feature because a customer should be able to check a product before he buys it, directly from the store.

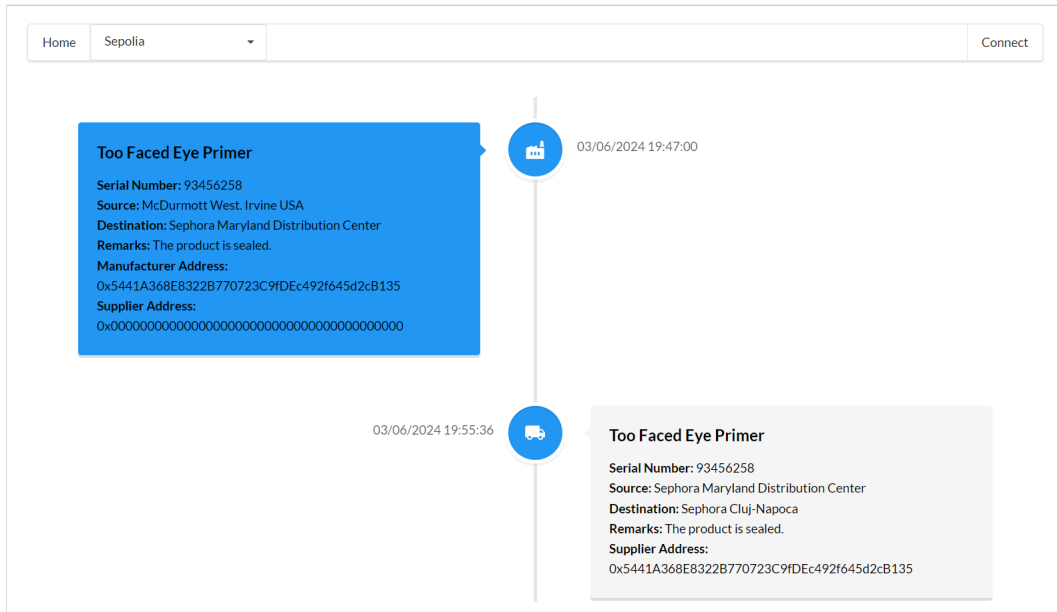


Figure 7.8: History of a product

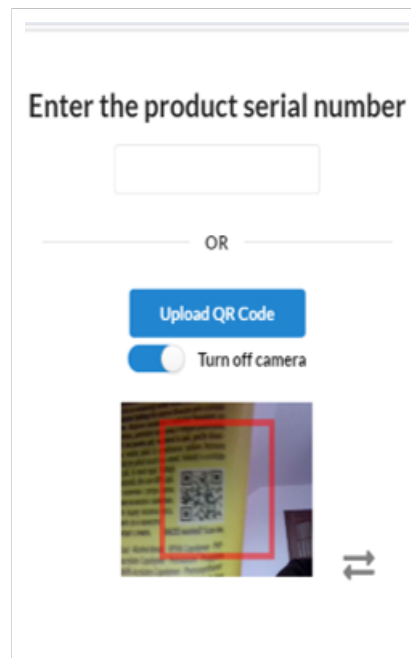


Figure 7.9: Accessing the application from a phone

Chapter 8. Conclusions

8.1. Accomplished objectives

The primary objective of this thesis was to improve the transparency and traceability of products in the supply chain by using blockchain technology. This objective was successfully achieved through the development of a blockchain-based system that securely records and verifies the history of products in a transparent and tamper-proof manner. By implementing this system, the thesis addressed the critical issue of ensuring product authenticity and traceability, which is crucial for combating counterfeiting and fraud in supply chains. The system allows all stakeholders to access reliable and immutable records of product origins, manufacturing processes, and distribution channels, thereby enhancing trust and accountability across the supply chain.

Furthermore, the thesis aimed to ensure the system's scalability, security, and usability, and these objectives were also accomplished. The system's architecture was designed to handle large volumes of transactions efficiently by relying on Ethereum's proof-of-stake mechanism. The integration of IPFS for off-chain storage, referenced via CIDs, was particularly effective in enhancing scalability. This approach reduced the financial costs associated with on-chain storage and significantly increased the transaction throughput.

In terms of gas consumption, storing product data directly on the blockchain resulted in high costs, with each transaction consuming approximately 220,000 gas. Given the maximum block size of 30,000,000 gas, this limited the number of transactions per block to about 136, resulting in a TPS of approximately 11. In contrast, using IPFS to store data and only recording CIDs on the blockchain reduced the gas cost per transaction to around 55,000. This allowed for approximately 545 transactions per block, resulting in a TPS of about 45. This marked improvement in TPS demonstrates the system's enhanced capability to manage high volumes of data and transactions while maintaining performance and responsiveness.

Usability was another key focus, and the system was designed with a user-friendly interface that simplifies interactions for various stakeholders. Additionally, the system's adaptability to different supply chain environments was a notable achievement, as it was designed to be flexible enough to accommodate various types of products and industry-specific requirements. This adaptability ensures that the system can be integrated into diverse supply chain contexts, providing a versatile solution that can address a wide range of transparency and traceability challenges.

The successful achievement of both primary and secondary objectives demonstrates the effectiveness and viability of the proposed solution. The blockchain-based system not only enhances transparency and traceability but also offers significant improvements in scalability, security, and usability, making it a robust and comprehensive solution to address modern supply chain challenges. The integration of IPFS played a crucial role

in these improvements, highlighting the importance of combining off-chain and on-chain storage solutions to optimize performance and cost-efficiency. These accomplishments suggest that the system is well-positioned for further advancements and can serve as a foundation for future research and development in the field of supply chain management.

8.2. Personal contributions

Extensive research and analysis were conducted to guide the development process and verify the system's effectiveness throughout the design of the thesis. This involved reviewing relevant literature, analyzing existing supply chain problems, and tracking advancements in blockchain technology. The personal research efforts led to a deeper understanding of the challenges and opportunities in supply chain management, which then directed the design and implementation of the proposed solution.

The most significant contribution to the project was the development of the smart contract that aligned with the initial objectives and established requirements. This smart contract was designed to ensure secure, transparent, and tamper-proof recording of product histories, addressing critical issues of authenticity and traceability in the supply chain. The development process required careful consideration of Ethereum's Proof-of-Stake mechanism and the optimization of gas usage to make the system both efficient and cost-effective.

Additionally, the integration of IPFS for off-chain storage was a significant personal contribution. Initially, the system stored data directly on the blockchain, resulting in high gas costs and limited scalability. By observing these challenges, the decision was made to integrate IPFS, which significantly reduced the costs of on-chain storage. This approach also improved the transaction throughput, increasing it from approximately 11 to 45 transactions per second. The integration of IPFS allowed the system to handle larger volumes of transactions efficiently, enhancing both performance and scalability.

8.3. Future improvements

The blockchain-based system developed in this thesis has demonstrated substantial advancements in supply chain transparency and traceability. However, there are several opportunities for future improvements that could further enhance its effectiveness and adaptability.

Integrating Internet of Things (IoT) devices into the system offers a major advantage by enabling real-time tracking and monitoring of products across the entire supply chain. Through IoT sensors strategically placed throughout the chain, valuable data can be collected, including precise location, temperature fluctuations, humidity levels, and various environmental conditions.

This data can then be securely recorded on the blockchain by using its immutable and transparent nature. By doing so, the integration of IoT and blockchain technologies enhances the system's capacity to deliver precise information. This real-time visibility significantly augments the overall efficiency and reliability of the supply chain management process.

Other improvements to the system could include the implementation of a dedicated registration page for manufacturers and suppliers. This page would allow new users to easily create accounts and register their businesses within the blockchain network.

By providing a structured and user-friendly registration interface, manufacturers and suppliers can efficiently enter their details, submit necessary documentation, and gain access to the system without extensive manual intervention.

Additionally, incorporating specific logic within the smart contracts to regulate user actions based on their roles will enhance system functionality and security. For instance, smart contracts can be programmed to allow manufacturers to only add products, while suppliers might be limited to update and create bundles of products. This role-based access control will ensure that each user type can perform only the actions relevant to their responsibilities, reducing the risk of errors and unauthorized activities.

These improvements will not only make the system more intuitive and user-friendly but also add an extra layer of security by restricting actions based on user roles. This approach will help maintain the integrity of the supply chain data and ensure that all participants operate within their designated capabilities.

Bibliography

- [1] C. A. Burgos and N. Wajzman, Eds., *Economic impact of counterfeiting in the clothing, cosmetics, and toy sectors in the EU*. European Union Intellectual Property Office, 2024.
- [2] European Commission, “Safety Gate: The EU Rapid Alert System for Dangerous Non-Food Products,” 2024, accessed on: 2024-03-25. [Online]. Available: <https://ec.europa.eu/safety-gate-alerts/screen/search?resetSearch=true>
- [3] Pfizer Inc., “2019 financial report,” 2019, accessed on: 2024-03-25. [Online]. Available: https://www.annualreports.com/HostedData/AnnualReportArchive/p/NYSE_PFE_2019.pdf
- [4] “Ethereum Average Gas Limit Chart,” <https://etherscan.io/chart/gaslimit> [Accessed 2024.03.10]. [Online]. Available: <https://etherscan.io/chart/gaslimit/>
- [5] “Ethereum Average Block Time Chart,” <https://etherscan.io/chart/blocktime> [Accessed 2024.03.10]. [Online]. Available: <https://etherscan.io/chart/blocktime/>
- [6] Dr. Gavin Wood. “Ethereum: A Secure Decentralised Generalised Transaction Ledger” Paris Version 705168a – 2024-03-04, [Accessed 2024.03.10]. [Online]. Available: <https://ethereum.github.io/yellowpaper/paper.pdf>
- [7] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*, first edition ed. O’Reilly Media, Inc., 2014.
- [8] S. Anwar, S. Anayat, S. Butt, S. Butt, and M. Saad, “Generation analysis of blockchain technology: Bitcoin and ethereum,” *International Journal of Information Engineering and Electronic Business*, vol. 12, pp. 30–39, 08 2020.
- [9] M. Alizadeh, K. Andersson, and O. Schelén, “Efficient decentralized data storage based on public blockchain and ipfs,” 12 2020, pp. 1–8.
- [10] M. D. Praveen, S. G. Totad, M. Rashinkar, R. Ostwal, S. Patil, and P. M. Hadapad, “Scalable blockchain architecture using off-chain ipfs for marks card validation,” *Procedia Computer Science*, vol. 215, pp. 370–379, 2022, 4th International Conference on Innovative Data Communication Technology and Application. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S187705092202110X>
- [11] P. Tuyls and L. Batina, “Rfid-tags for anti-counterfeiting,” 02 2006, pp. 115–131.
- [12] E. Daoud, D. Vu Nguyen Hai, H. Nguyen, and M. Gaedke, “Improving fake product detection using ai- based technology,” 04 2020.

-
- [13] A. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly, 2018. [Online]. Available: <https://books.google.ro/books?id=SedSMQAACAAJ>
 - [14] X. Zhu, "Consensus algorithms in blockchain : A survey to create decision trees for blockchain applications," 2023. [Online]. Available: <https://www.diva-portal.org/smash/get/diva2:1772618/FULLTEXT01.pdf>
 - [15] B. Mohanta, S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," 10 2018.
 - [16] D.-H. Shin, J. Jung, and B.-H. Chang, "The psychology behind qr codes: User experience perspective," *Computers in Human Behavior*, vol. 28, no. 4, pp. 1417–1426, 2012. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0747563212000702>
 - [17] D. Sharma, "A review of qr code structure for encryption and decryption process," 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:246245497>
 - [18] W. Abdulhakeem and S. Dugga, "Enhancing user experience using mobile qr-code application," *International Journal of Computer and Information Technology*, vol. 03, pp. 1310–1315, 11 2014.
 - [19] (2024) Redux documentation. Accessed: 2024-05-13. [Online]. Available: <https://redux.js.org/>
 - [20] (2024) Ethers.js documentation. Accessed: 2024-05-13. [Online]. Available: <https://docs.ethers.org/v5/>
 - [21] (2024) Metamask documentation. Accessed: 2024-05-14. [Online]. Available: <https://docs.metamask.io/>
 - [22] (2024) Infura documentation. Accessed: 2024-05-14. [Online]. Available: <https://www.infura.io/>
 - [23] Chainlink. (2023) How To Get Sepolia Testnet ETH [Accessed 2024.05.12]. [Online]. Available: <https://blog.chain.link/sepolia-eth/#:~:text=Sepolia%20is%20the%20recommended%20default,2021%20by%20Ethereum%20core%20developers.>
 - [24] GeeksforGeeks. (2023) What is Goerli Testnet? [Accessed 2024-05-12]. [Online]. Available: <https://www.geeksforgeeks.org/what-is-goerli-testnet/>
 - [25] Hardhat, "Hardhat: Ethereum development environment," 2024, Accessed: 2024-05-12. [Online]. Available: <https://hardhat.org/>
 - [26] (2024) Solidity mapping types. Accessed: 2024-05-15. [Online]. Available: <https://docs.soliditylang.org/en/v0.8.7/types.html#mapping-types>
 - [27] Kaan Kaçar, "Solidity events explained," 2023, Accessed on: 2024-05-19. [Online]. Available: <https://medium.com/coinmonks/solidity-events-explained-82dc9104bc62#:~:text=In%20Solidity%2C%20events%20are%20a,it%20transparent%20and%20easily%20accessible.>

- [28] (2024) Remix IDE. Accessed: 2024-05-20. [Online]. Available: <https://remix.ethereum.org/>
- [29] (2024) Pinata. Accessed: 2024-05-24. [Online]. Available: <https://docs.pinata.cloud/introduction>
- [30] “Sepolia Average Block Time,” <https://eth-sepolia.blockscout.com/> [Accessed 2024.06.02]. [Online]. Available: <https://eth-sepolia.blockscout.com/>.
- [31] (2024) On Block Sizes, Gas Limits and Scalability. Accessed: 2024-05-24. [Online]. Available: <https://ethresear.ch/t/on-block-sizes-gas-limits-and-scalability/18444>
- [32] Georgiana Grațîela Stănea. (2024) Project Source Code [Accessed 2024-06-03]. [Online]. Available: <https://github.com/georgianastanea/fake-product-identification.git>

Appendix A. *ProductTracker* Smart Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ProductTracker {
    mapping(uint => string) products;
    mapping(uint => uint[]) complexProducts;

    event ProductTracker__AddProduct(
        uint serialNumber,
        string cid,
        uint timestamp
    );

    event ProductTracker__UpdateProduct(
        uint serialNumber,
        string cid,
        uint timestamp
    );

    event ProductTracker__AddComplexProduct(
        uint serialNumber,
        uint[] containingProducts,
        uint timestamp
    );

    event ProductTracker__UpdateComplexProduct(
        uint serialNumber,
        uint[] containingProducts,
        uint timestamp
    );

    function addProduct(uint _serialNumber, string memory _cid) public {
        require(!productExists(_serialNumber), "Product already exists");
        products[_serialNumber] = _cid;
        emit ProductTracker__AddProduct(_serialNumber, _cid, block.timestamp);
    }

    function updateProduct(uint _serialNumber, string memory _cid) public {
        require(productExists(_serialNumber), "Product not found");
        products[_serialNumber] = _cid;
        emit ProductTracker__UpdateProduct(_serialNumber, _cid, block.timestamp);
    }

    function addComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
        require(!productExists(_serialNumber), "Product already exists");
        for (uint i = 0; i < _containingProducts.length; i++) {
            require(productExists(_containingProducts[i]), "Component product not found");
        }
    }
}
```

```

    }
    complexProducts[_serialNumber] = _containingProducts;
    emit ProductTracker__AddComplexProduct(
        _serialNumber,
        _containingProducts,
        block.timestamp);
}

function updateComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
    require(productExists(_serialNumber), "Complex product not found");
    for (uint i = 0; i < _containingProducts.length; i++) {
        require(productExists(_containingProducts[i]), "Component product not found");
    }
    complexProducts[_serialNumber] = _containingProducts;
    emit ProductTracker__UpdateComplexProduct(
        _serialNumber,
        _containingProducts,
        block.timestamp);
}

function getProduct(uint _serialNumber) public view returns (string memory) {
    return products[_serialNumber];
}

function getComplexProduct(uint _serialNumber) public view returns (uint[] memory) {
    return complexProducts[_serialNumber];
}

function productExists(uint _serialNumber) public view returns (bool) {
    return bytes(products[_serialNumber]).length > 0;
}
}

```

Appendix B. *ProductTracker* Smart Contract without IPFS

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ProductTracker {

    mapping(uint => Product) products;
    mapping(uint => bool) productExists;
    mapping(uint => uint[]) complexProducts;

    struct Product {
        uint serialNumber;
        uint timestamp;
        string name;
        string source;
        string destination;
        string remarks;
        address manufacturer;
        address supplier;
    }

    event ProductTracker__AddProduct(
        uint serialNumber,
        uint timestamp,
        string name,
        string source,
        string destination,
        string remarks,
        address manufacturer,
        address supplier
    );

    event ProductTracker__UpdateProduct(
        uint serialNumber,
        uint timestamp,
        string name,
        string source,
        string destination,
        string remarks,
        address manufacturer,
        address supplier
    );

    event ProductTracker__AddComplexProduct(
        uint serialNumber,
        uint[] containingProducts,
        uint timestamp
    );
};
```

```

event ProductTracker__UpdateComplexProduct(
    uint serialNumber,
    uint[] containingProducts,
    uint timestamp
);

function addProduct(
    uint _serialNumber,
    string memory _name,
    string memory _source,
    string memory _destination,
    string memory _remarks
) public {
    require(!productExists[_serialNumber], "Product already exists");

    products[_serialNumber] = Product(
        _serialNumber,
        block.timestamp,
        _name,
        _source,
        _destination,
        _remarks,
        msg.sender,
        address(0)
    );
    productExists[_serialNumber] = true;

    emit ProductTracker__AddProduct(
        _serialNumber,
        block.timestamp,
        _name,
        _source,
        _destination,
        _remarks,
        msg.sender,
        address(0)
    );
}

function updateProduct(
    uint _serialNumber,
    string memory _name,
    string memory _source,
    string memory _destination,
    string memory _remarks
) public {
    require(productExists[_serialNumber], "Product not found");

    Product storage product = products[_serialNumber];
    product.name = _name;
    product.source = _source;
    product.destination = _destination;
    product.remarks = _remarks;
    product.supplier = msg.sender;

    emit ProductTracker__UpdateProduct(
        _serialNumber,

```



```

        block.timestamp,
        _name,
        _source,
        _destination,
        _remarks,
        product.manufacturer,
        msg.sender
    );
}

function addComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
    require(!productExists[_serialNumber], "Product already exists");
    for (uint i = 0; i < _containingProducts.length; i++) {
        require(productExists[_containingProducts[i]], "Component product not found");
    }
    complexProducts[_serialNumber] = _containingProducts;
    productExists[_serialNumber] = true;
    emit ProductTracker__AddComplexProduct(_serialNumber, _containingProducts, block.timestamp);
}

function updateComplexProduct(uint _serialNumber, uint[] memory _containingProducts) public {
    require(productExists[_serialNumber], "Complex product not found");
    for (uint i = 0; i < _containingProducts.length; i++) {
        require(productExists[_containingProducts[i]], "Component product not found");
    }
    complexProducts[_serialNumber] = _containingProducts;
    emit ProductTracker__UpdateComplexProduct(
        _serialNumber, _containingProducts, block.timestamp);
}

function getProduct(uint _productId) public view returns (
    uint,
    uint,
    string memory,
    string memory,
    string memory,
    string memory,
    address,
    address
) {
    require(productExists[_productId], "Product not found");
    Product storage product = products[_productId];
    return (
        product.serialNumber,
        product.timestamp,
        product.name,
        product.source,
        product.destination,
        product.remarks,
        product.manufacturer,
        product.supplier
    );
}

function getTimestamp(uint _productId) public view returns (uint) {
    require(productExists[_productId], "Product not found");
    return products[_productId].timestamp;
}

```

```

function getSerialNumber(uint _productId) public view returns (uint) {
    require(productExists[_productId], "Product not found");
    return products[_productId].serialNumber;
}

function getName(uint _productId) public view returns (string memory) {
    require(productExists[_productId], "Product not found");
    return products[_productId].name;
}

function getSource(uint _productId) public view returns (string memory) {
    require(productExists[_productId], "Product not found");
    return products[_productId].source;
}

function getDestination(uint _productId) public view returns (string memory) {
    require(productExists[_productId], "Product not found");
    return products[_productId].destination;
}

function getRemarks(uint _productId) public view returns (string memory) {
    require(productExists[_productId], "Product not found");
    return products[_productId].remarks;
}

function getManufacturer(uint _productId) public view returns (address) {
    require(productExists[_productId], "Product not found");
    return products[_productId].manufacturer;
}

function getSupplier(uint _productId) public view returns (address) {
    require(productExists[_productId], "Product not found");
    return products[_productId].supplier;
}

function getComplexProduct(uint _serialNumber) public view returns (uint[] memory) {
    require(productExists[_serialNumber], "Complex product not found");
    return complexProducts[_serialNumber];
}
}

```

Appendix C. Published Papers

- **G. G. Stănea**, O. A. Marin, C. Antal, M. Antal, , "Supply Chain Management using Distributed Ledgers for Data Immutability and Integrity", submitted for IEEE 20th International Conference on Intelligent Computer Communication and Processing, 2024

Supply Chain Management using Distributed Ledgers for Data Immutability and Integrity

Georgiana-Grațîela Stănea, Oana Andreea Marin, Claudia Antal, Marcel Antal

*Computer Science Department
Technical University of Cluj-
Napoca Cluj-Napoca, Romania*

georgiana.stanea@student.utcluj.ro, {[oana.marin](mailto:oana.marin@utcluj.ro), [claudia.antal](mailto:claudia.antal@utcluj.ro), [marcel.antal](mailto:marcel.antal@utcluj.ro)}@cs.utcluj.ro

Abstract— This paper addresses the problem of data corruption and integrity in the case of supply chains that often face problems such as lack of transparency and data corruption that led to counterfeit products or products with misleading or incorrect specification or expiration dates, by proposing a supply chain management system implemented on a distributed ledger. The main properties of blockchain are data immutability and integrity, offering transparency and traceability for data. Considering this, we propose an Ethereum-based solution, where the main operations needed for tracking products are implemented using smart contracts. The contracts are designed to abstract the possibility of having composite products composed of basic products, allowing the storage of entire supply chains with products and their components, tracing back to the raw materials. Furthermore, to enhance the system scalability and lower the costs, we integrate the blockchain system with Inter Planetary File System (IPFS), storing data off-chain, but locating it using a Content Identifier, that relies on the data hash to locate it in the distributed system, assuring data integrity and identifying corrupt data. Finally, the system is implemented in Ethereum blockchain, testing it on a test-net with similar parameters as the public network, showing a throughput of 11 operation per second for adding products, at a cost of 2.5\$ each. By integrating the system with IPFS, we double the throughput and lower the costs by more than 50%, achieving a throughput of 26 TPS for adding products at a cost of 1.13\$ each.

Keywords – *Ethereum Blockchain, Supply Chain Management, IPFS, smart contracts.*

I. INTRODUCTION

Counterfeit goods are a widespread issue affecting numerous sectors, from electronics and automotive components to consumer products and pharmaceuticals. This problem results in manufacturers and suppliers losing billions of dollars each year. However, the risks consumers are exposed to are often more severe. Fake auto parts or consumer goods that malfunction can lead to overheating or fires, and counterfeit medications contribute to over a million deaths annually.

In the domain of supply chain management, the detection and prevention of counterfeit products represents a significant and complex challenge. The lack of transparency in traditional supply chains led to issues such as service redundancy, poor coordination among departments, and lack of standardization, contributing to the extension of counterfeit goods.

The counterfeit cosmetics market worldwide has been experiencing a notable increase, mirroring global trends in the widespread distribution of fake makeup products. According to a report by the European Union Intellectual Property Office (EUIPO) [1], the cosmetics sector in the EU loses approximately €3 billion annually due to the presence of counterfeit products. This figure represents a significant

portion of the market, with counterfeit goods not only affecting consumer confidence but also causing substantial financial losses for genuine brands.

The impact of using counterfeit cosmetics can be severe, producing health risks due to the presence of hazardous substances. European authorities have reported instances of counterfeit products containing dangerous levels of heavy metals such as lead and mercury, alongside carcinogenic substances like arsenic and cadmium. These findings underscore the health hazards posed by such products. The European Commission's rapid alert system for dangerous non-food products has consistently highlighted the issue, with numerous alerts issued for counterfeit cosmetics that fail to comply with EU health and safety standards [2].

Challenges also arise for legitimate manufacturers, impacting them in various ways including reduced sales, the expenses associated with brand protection, bad reputation, and the financial burden of disposing counterfeit goods and legal actions against counterfeiters or individuals affected by fake products. These issues are often referenced in a broad sense within corporate documentation. For instance, Pfizer, one of the top five global pharmaceutical firms, acknowledged the issue of counterfeiting in its 2019 annual financial statement. Within this financial document, the company dedicates a section to counterfeit products [3], offering an overview of the challenges encountered and detailing the measures undertaken to combat these issues.

Customers require credible ways of verifying the authenticity of the products they are interested in without being exposed to unwanted side-effects. The same applies to brands and manufacturers.

This paper addresses the issue of supply chain transparency to avoid counterfeits by proposing a supply chain management system based on blockchain technology to enhance the transparency and trust across the entire supply chain, ensuring that all parties involved have access to reliable information regarding product authenticity. By relying on the immutability of blockchain technology, the proposed system tracks the components of the products and the products themselves during their lifecycle. Furthermore, to enhance the scalability of the system, it is integrated with IPFS [24] to store large data while preserving the integrity of the data at lower storage and processing cost. An experimental prototype is developed and implemented to show the validity of the approach and to evaluate its throughput.

The rest of the paper is structured as follows: Section II shows related work, Section III presents the distributed architecture of the proposed solution, Section IV presents the operation of

the main use-cases, Section V presents results obtained in a test network configured with the same parameters as the public Ethereum network, while Section VI concludes the paper.

II. RELATED WORK

Supply chains management represents a complex scenario that involves multiple types of participants, all coordinating tasks and exchanging information and goods. The concept of Supply Chain Management (SCM) was introduced in the 2000s, enabling firms to strategically manage the flow of goods from raw materials to final products. Different types of supply chains have been identified based on their functions and goals:

- **Traditional Supply Chains:** These focus on the linear flow of products from suppliers to manufacturers, and then to distributors and retailers, ending with consumers. The main goal is efficiency and cost-effectiveness.
- **Green Supply Chains:** Emphasizing sustainability, these chains aim to minimize the environmental impact. This involves eco-friendly product and process designs as well as sustainable manufacturing practices.
- **Global Supply Chains:** These involve cross-border transactions and manage business relationships on an international scale. The complexity increases with the involvement of multiple countries, which requires careful management of logistics, legal compliance, and cultural differences.

In the context of supply chain management, blockchain technology supports various functions, including enhancing data transparency, ensuring data immutability, and improving trust among participants. By leveraging smart contracts, digital signatures, and cryptographic hash functions, blockchain can significantly enhance the efficiency and integrity of supply chains. We identify multiple domains of applying blockchain for supply-chain management, such as food industry, pharmaceuticals, medical systems and digital sector.

To begin with, in the food industry, a solution to track food all the way from farmer to the store using a permissioned blockchain is proposed in [4]. Blockchain has been used to trace the origin of food products, ensuring safety and quality by providing detailed information about each stage of the supply chain [5]. Another solution for fruit and vegetable tracing during the supply chain and distribution is proposed in [6]. The data for tracing products is introduced either manually by retailers or fed automatically into the system using IoT devices. The details regarding products are stored in a relation database, while the data hash is stored on blockchain to assure data immutability. QR codes are used to retrieve product related data by entities manipulating the products during their distribution. A similar system is proposed in [7], where a blockchain system is designed to wine supply chain traceability. The system stores on blockchain information regarding grape harvesting, processing, wine manufacturing and wine bottling, assigning unique codes for each bottle of wine. Using the code assigned to each bottle, the customers can use an online platform to check the wine traceability and make sure it is valid. As soon

as a bottle is sold, it is marked on blockchain as sold so it cannot be cloned by counterfeit products.

Another domain where blockchain is applied in supply chain management is the pharmaceutical domain. Blockchain helps in combating counterfeit drugs by providing an immutable record of drug production and distribution, ensuring that consumers receive authentic products [8]. A platform for Covid19 vaccine traceability is proposed in [9], the authors leveraging on the immutability property of blockchain systems to make sure Covid19 vaccines cannot be counterfeit, and that during their distribution the safety rules are applied (i.e. ambient temperature for distribution). Furthermore, the system traces vaccines from manufacturing to doctors and patients, storing their trace and updates. Vaccines are identified through QR codes. Data is stored in IPFS to lower costs for utilizing blockchain when storing large volumes of data. A similar system is proposed in [10], where authors use blockchain and IPFS to store medication-related data, tracing their distribution from factories to hospitals and pharmacies. Each lot of medicine is stored on blockchain, being associated to a QR code, that when scanned offers information about manufacturing and distribution as well as possible problems identified with that lot. Furthermore, using blockchain assures the end-users that the medicine is not counterfeit.

Furthermore, regarding the medical system, the authors of [11] propose a system for supply chain management of blood donations in the medical systems. The proposed system uses blockchain to ensure the traceability of each unit of blood collected from the moment in which it is taken from the donor, and until it ends up being administered to the patient. The system uses an Ethereum network for information storage, and smart contracts that store information about the donor, blood information, blood tests, storage conditions and finally the patient information. Bar codes are used to link physical blood samples to information stored on blockchain. The authors of [12] propose a blockchain-based system for supervising the waiting lists for transplants. The proposed system designs an information sharing scheme between medical parties, as well as distributed file systems for large volume of data storage, integrated with blockchain for data integrity and traceability. The experimental results illustrate the system's trust and scalability while reducing the possibility of stealing or altering medical data.

Finally, various solutions for supply chains in digital sectors are applied. The authors of [13] propose a blockchain-based system for issuing and tracking Renewable Energy Certificates (RECs), offering transparency and traceability, as opposed to classical centralized systems. Furthermore, the authors of [14] present a blockchain-based system for traceability of photos on the Internet called PhoTrace, aiming to combat copyright violations, to determine the author, the camera with which it was captured, as well as possible editions. When the photo is taken, a digital certificate is automatically generated, containing the camera details as a signature. This certificate together with the image itself are stored on the blockchain. At any editing of the image, a new digital certificate is generated which is attached to the already existing certificate. The author of the image can sell it online, authorizing the usage of the copyright of the image and the

buyer can check the information associated with an image before purchasing it.

The current work is built upon the related work presented, proposing a distributed architecture for implementing a generic supply chain management system based on distributed ledger and IPFS for scalability, aiming to assure data immutability and integrity while tracking the products lifecycle. As opposed to the state of the art, the proposed system is capable of tracking generic items that can be composed of sub-items, similar to a composite design pattern [15], allowing storing and tracing information related to composed products and tracing back each component to the original manufacturer. Furthermore, IPFS is integrated with blockchain to increase system scalability by storing large volumes of data off-chain, while maintaining data integrity using hash values stored on-chain. Finally, the paper contributions are:

- Propose a distributed architecture and Smart Contracts based on Ethereum distributed ledger for abstracting the concepts involved supply chain management and allowing to represent composite products and tracking the changes during their lifecycle.
- Implement an experimental prototype in Ethereum to validate the proposed architecture and integrating with IPFS to increase the system scalability and throughput
- Evaluate the proposed system and comparing the two approaches, with and without IPFS on an experimental setup with parameters similar to public Ethereum network to show the system applicability in the real world

III. DISTRIBUTED SUPPLY MANAGEMENT SYSTEM

The system uses blockchain technology to build a safe, decentralized network for tracking product details, decreasing counterfeiting, and improving transparency across supply chains. By recording product information on a blockchain, manufacturers and providers have an incorruptible, steady, and private way to uphold data integrity. Customers profit from this system by being able to access the full supply chain record of items, enabling them to confirm authenticity and guarantee they are not buying counterfeit products.

The conceptual architecture of the proposed solution is presented in Figure 1. The system is composed of three major components: A) a set of client applications used by regular clients, B) the peer-to-peer blockchain network, storing and processing the data and C) the IPFS system, storing large volumes of data.

A. Client Application for Interacting with the Distributed Ledger

The web application serves as the primary interface for users to interact with the system. It is designed to provide a user-friendly and responsive experience for manufacturers, suppliers, and customers. The application employs Ethers.js

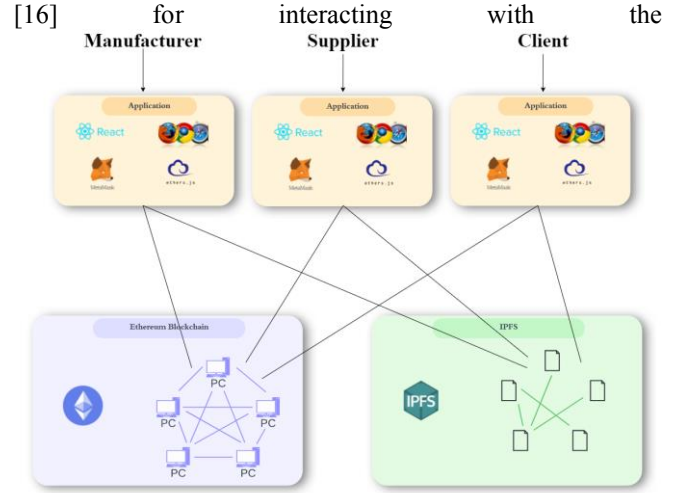


Figure 1. Conceptual system architecture

Ethereum blockchain [17], facilitating secure and seamless transactions. Ethers.js simplifies interactions with smart contracts, enabling users to perform actions such as registering new products and verifying product authenticity directly from the web interface. The proposed system is a true peer-to-peer system, each client application running in the browser and connecting directly to the blockchain network.

Security related requirements, such as authentication and authorization are implemented using MetaMask [18], that is integrated into the web application to handle user authentication and wallet management. It provides a secure interface for managing Ethereum accounts, signing transactions, and interacting with decentralized applications, ensuring that users' private keys are securely stored and never exposed to the application. Furthermore, Infura [19] is utilized as a fallback mechanism to connect the web application to the Ethereum network when no provider is injected into the web browser via MetaMask. This ensures the application remains operational even without MetaMask, providing reliable and scalable access to the Ethereum blockchain.

B. Distributed ledger for supply chain management

Another important component of the system is the Ethereum blockchain, which supports various environments for executing transactions, including the Sepolia [20] and Goerli [21] testnets, as well as a local blockchain deployed with Hardhat [22]. The Sepolia Testnet replicates Ethereum mainnet conditions with fast sync times and low storage requirements, while the Goerli Testnet offers a stable platform for testing new features with a Proof-of-Authority consensus. The local blockchain with Hardhat provides a versatile development environment for efficient testing and debugging.

Smart contracts form the core of our distributed supply management system, automating and securing processes such as product registration, updates, and verification. These contracts are written in Solidity and define the rules and workflows for managing product information on the blockchain.

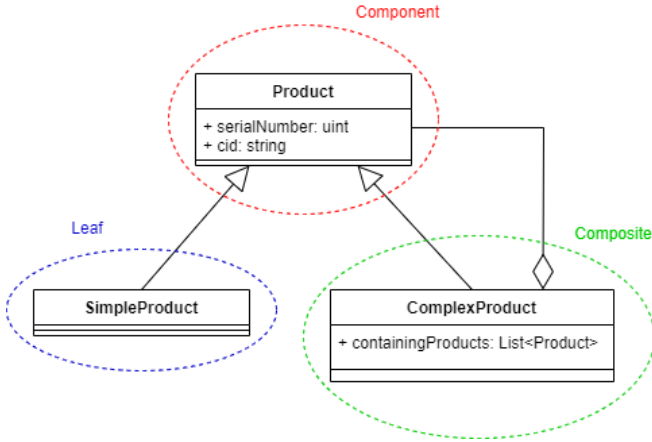


Figure 2. Managing products

One of the key components of this system is the implementation of complex products using a structure similar to the Composite Design Pattern [23], as presented in Figure 2. This design pattern is particularly well-suited for scenarios where individual objects and compositions of objects need to be treated uniformly.

In our implementation, the contract is designed to manage both simple and complex products within the supply chain. Simple products are tracked individually using unique serial numbers and associated metadata. Conversely, complex products are represented as compositions of these simple products. This dual structure allows individual products (leaves) and composite products (nodes) to be manipulated through a consistent interface.

The application of this structural pattern within our blockchain-based supply chain management system significantly enhances its robustness and scalability. By enabling uniform treatment of simple and complex products, our system ensures seamless interaction and manipulation of products throughout the supply chain.

In the initial version of the application, all product details were stored directly within the smart contracts. A partial implementation of this version can be visualized in Figure 3. This approach involved creating a product structure that included various attributes such as serial number, name, source, destination, remarks, and the addresses of the manufacturer and supplier. The product details were stored in mappings, allowing efficient querying and retrieval of product information.

```

1  MAP (uint serialNumber, Product product) _products
2  MAP (uint serialNumber, bool exists) _productExists
3  MAP (uint serialNumber, containingProducts[]) _complexProducts
4
5  Struct Product:
6      uint serialNumber
7      uint timestamp
8      string name
9      string source
10     string destination
11     string remarks
12     address manufacturer
13     address supplier
14
15  Events: AddProduct, UpdateProduct, AddComplexProduct, UpdateComplexProduct

```

Figure 3. Initial implementation of the smart contract

The function for adding a product, for instance, involved checking if the product already existed, and if not, storing the product details and emitting an event to record the transaction. Similarly, updating a product required verifying

its existence, modifying the relevant attributes, and emitting an update event. Complex products composed of multiple components were managed by storing arrays of product serial numbers, ensuring that each component's provenance was maintained.

C. IPFS for storing large volume of data

While this approach demonstrated the feasibility of using blockchain for product tracking, it also highlighted several significant challenges. One major issue was the high gas costs associated with storing extensive product details directly on-chain. Each additional piece of data increased the gas required for transactions, making the process expensive and less viable for large-scale deployments. As the number of products and their associated updates grew, the blockchain became more congested, leading to scalability issues. The large volume of data stored on-chain not only resulted in higher costs but also slowed down transaction times and increased the complexity of data retrieval.

To address these challenges, we transitioned to a solution that uses the InterPlanetary File System [24] for decentralized off-chain storage. Its foundational principle of content addressing ensures data integrity by assigning each piece of content a unique cryptographic hash derived from its contents. Any alteration to the data results in a different hash, making tampering immediately detectable. This hash serves as a secure identifier, effectively verifying the authenticity and integrity of files stored and shared within the network. Unlike traditional centralized servers, IPFS employs a distributed architecture where data is broken down into smaller chunks, each uniquely identified by its hash. These chunks are then distributed across a network of interconnected nodes, enabling efficient retrieval through parallel fetching from multiple sources.

By storing detailed product information on IPFS and only recording content identifiers on the blockchain, the gas consumption was significantly reduced, and scalability improved. This approach minimized the amount of data stored on-chain, lowered transaction costs, and enhanced transaction throughput. A partial implementation of the final version of the smart contract can be seen in Figure 4.

```

1  Smart Contract: ProductTracker
2
3  MAP (uint serialNumber, string cid) _products
4  MAP (uint serialNumber, containingProducts[]) _complexProducts
5
6  Events: AddProduct, UpdateProduct, AddComplexProduct, UpdateComplexProduct
7  Functions: addProduct, updateProduct, addComplexProduct, updateComplexProduct,
             getProduct, getComplexProduct, productExists

```

Figure 4. Implementation of the smart contract after integrating IPFS

IV. USE CASES FOR SUPPLY CHAIN MANAGEMENT

Several key use cases that are crucial for comprehensive supply chain management will be presented below. These use cases demonstrate how the system operates in real-world scenarios, ensuring that all stakeholders can effectively manage and verify product information.

1. Manufacturer Adds a Product

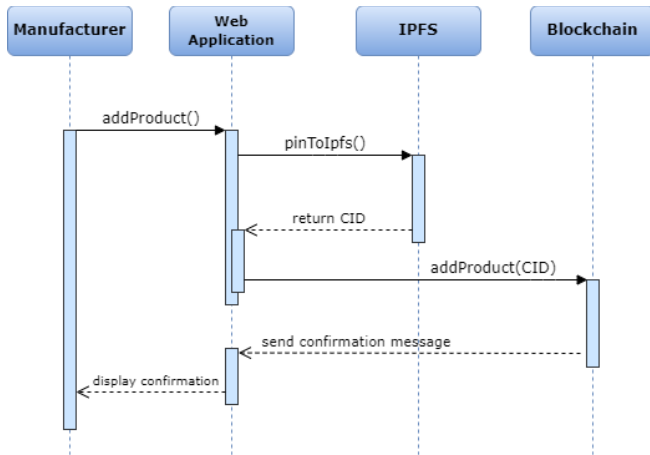


Figure 5. The process of adding a product

As presented in Figure 5, manufacturers can add new products to the blockchain, initiating the process through the web application. The manufacturer enters the product details, which are then stored on IPFS, generating a unique Content Identifier (CID). The CID is generated by combining the cryptographic hash of the content with additional information of the content, making it unique for a specific content information. This CID is subsequently recorded on the Ethereum blockchain via the *addProduct()* function in the smart contract. The blockchain records the CID, ensuring that the product information is immutable and verifiable. This process guarantees that all product details are securely stored and easily accessible for future reference.

2. Supplier Updates a Product

This use case involves the supplier updating existing product information to ensure it remains current and accurate. The process begins when the supplier initiates an update request via the web application. The web application first interacts with the blockchain to retrieve the current Content Identifier associated with the product's serial

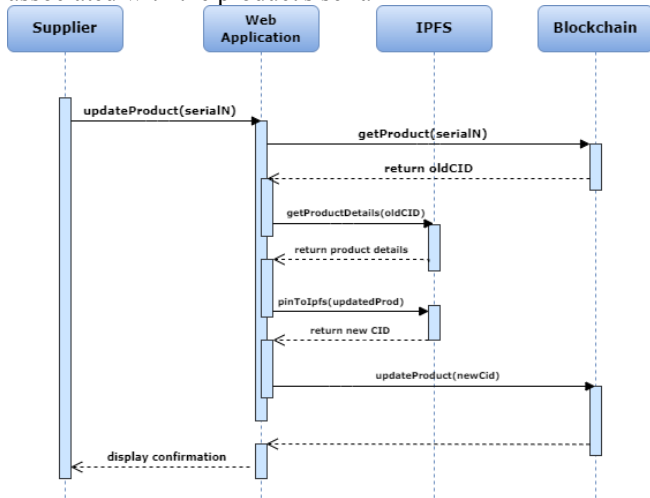


Figure 6. The process of updating a product

number. This CID acts as a unique reference to the product's stored data.

Once the current CID is obtained, the web application requests detailed product information from the InterPlanetary File System, using this CID to fetch the

necessary data. The supplier then updates the product information through the web application interface. The revised data is subsequently stored on IPFS, resulting in a new CID that reflects the updated product details.

Following this, the web application updates the blockchain with the new CID, ensuring that the updated product information is recorded in a secure and immutable manner. This updated CID replaces the old one, thereby maintaining a continuous and verifiable history of the product data. The process concludes with the web application providing confirmation to the supplier, indicating that the product update has been successfully completed. The described scenario can be visualized in Figure 6.

3. Customer Searches for a Product

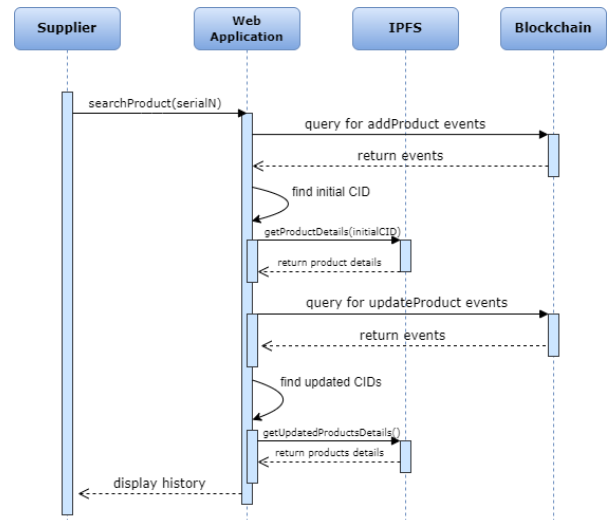


Figure 7. The process of searching for a product

Customers can verify the authenticity and trace the history of a product by searching for it using the web application, as presented in Figure 7. The process begins when the customer initiates a search by entering the product's serial number into the web application or scans a QR code using his device. The application then queries the blockchain for events associated with the addition of the product. The blockchain responds with these events, which contain the initial Content Identifier linked to the product.

With the initial CID obtained, the web application retrieves the detailed product information stored on IPFS. This information includes all data initially recorded about the product, ensuring that the customer can see the original details as registered at the point of manufacture or first entry into the system.

The web application also queries the blockchain for any subsequent events that updated the product's information. These update events contain new CIDs that correspond to changes made to the product data over time. By obtaining these updated CIDs, the application can fetch the corresponding updated product details from IPFS.

Finally, the web application compiles all retrieved data to provide the customer with a complete history of the product, from its initial registration to the most recent updates. This information is displayed in an intuitive format, allowing the customer to verify the product's authenticity and trace its entire history.

V. EXPERIMENTS AND VALIDATION

A. Experimental Setup

A test Ethereum network was configured with the same parameters as the public network, as shown in Table 1.

Table 1. Network setup parameters

Parameter	Value
Block time	12.05 seconds
Gas size	15 million gas
Consensus Mechanism	Proof of stake

The system was configured to evaluate its performance and scalability in two scenarios: one without IPFS for data storage, and the other with IPFS integrated. This setup allows for a comparative analysis of the system's efficiency and scalability under different configurations.

In the first scenario, all product information is stored directly on the Ethereum blockchain. This approach involves embedding detailed data within the smart contracts themselves. Each product entry, including its attributes, is recorded on-chain. As a result, every transaction related to product registration or update must handle large data payloads. This method aims to highlight the challenges associated with high gas consumption and the potential for increased costs when the blockchain is used as the sole data storage medium.

In the second scenario, detailed product information is stored on IPFS, and only the content identifier (CID) is stored on the Ethereum blockchain. This method uses the IPFS's decentralized storage capabilities to handle the bulk of the data, while the blockchain only keeps a reference to this data. The CID, a unique hash representing the product information on IPFS, ensures data integrity and immutability. This approach aims to demonstrate how offloading data storage to IPFS can reduce the gas costs associated with on-chain transactions, thus improving overall system efficiency and scalability.

To evaluate the system's scalability, several key metrics are used. The first one is Transactions per Second (TPS), which measures the number of transactions the network can process within a second.

$$TPS = \frac{blockGas}{transactionGas} * \frac{1}{blockTime} \quad (1)$$

It provides an indication of the system's capacity to handle high transaction volumes. The second metric is gas consumption, which is the amount of gas used for various operations. By analyzing gas consumption, we can assess the economic viability of the system under different configurations. These metrics provide a comprehensive view of the system's performance and cost-efficiency, offering insights into the trade-offs and benefits of each approach.

B. Experimental Results for Scalability Evaluation

The proposed system was evaluated in terms of scalability performed on multiple types of transaction and gas consumption.

When storing files directly on the blockchain, each transaction involves embedding detailed product information within the smart contracts. This approach leads to significantly higher gas consumption due to the large data payloads. For example, registering a new product with comprehensive details involves multiple "SSTORE" operations, each contributing to the overall gas cost. The

results showed that storing a typical product's details directly on the blockchain required approximately 210,000 gas units per transaction.

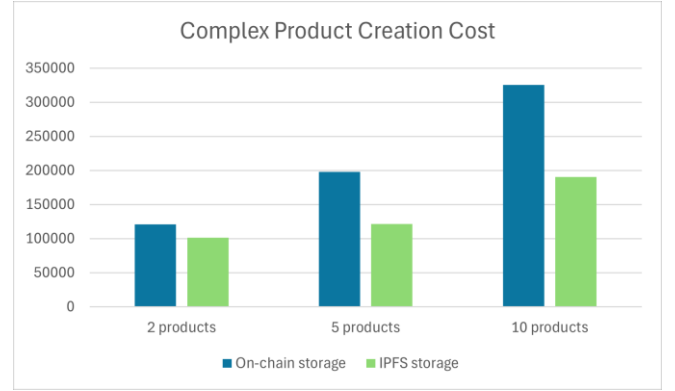


Figure 8. ComplexProduct Creation Cost in gas

The higher gas fees associated with these transactions make this method less economically viable for large-scale deployments. Additionally, the increased data volume per transaction resulted in a lower number of transactions per second, averaging around 11 TPS, indicating reduced scalability.

Table 2. Comparison of Gas Costs and TPS for On-chain and IPFS Storage Scenarios

Scenario	Operation	Gas Consumption	TPS
On-chain storage	Contract Deployment	1,755,313	1.42
	Add product	215,310	11.61
	Update product	91,165	27.42
	Add complex product	121,047	20.65
	Update complex product	43,944	56.89
IPFS storage	Contract Deployment	921,408	2.71
	Add product	93,970	26.60
	Update product	39,794	62.82
	Add complex product	101,220	24.69
	Update complex product	30,048	83.20

In the scenario where IPFS was used for storage, only the content identifier (CID) of the product information was stored on the blockchain, while the detailed data was stored off-chain on IPFS. This configuration demonstrated a substantial reduction in gas consumption per transaction since the on-chain data was minimized to the CID. The reduced gas usage resulted in lower transaction costs, with transactions requiring approximately 55,000 gas units. This approach also proved more economically viable, as the reduced gas costs made it feasible to scale the system for large volumes of transactions. Moreover, the system could handle a higher TPS, averaging around 45 TPS, indicating improved scalability. Storing data on IPFS ensured that the blockchain remained less congested, further enhancing system performance.

The analysis reveals significant differences in storage consumption, gas costs, and scalability between storing product information directly on the blockchain versus using IPFS.

IPFS consistently demonstrates lower storage requirements. It reduces storage needs by approximately 47.5% for contract deployment, 56.4% for adding and updating products, 16.4% for adding complex products, and 31.6% for updating complex products. This resource efficiency leads to substantial cost savings and enhanced scalability.

Regarding gas costs, IPFS shows a clear advantage. For creating complex products, IPFS storage requires significantly less gas than on-chain storage. This efficiency increases with the number of products, making IPFS a more cost-effective solution for large-scale applications.

The Transactions Per Second analysis further underscores the scalability benefits of using IPFS. Direct on-chain storage supports around 11 TPS, limited by high gas consumption per transaction. In contrast, using IPFS allows for approximately 45 TPS, thanks to the reduced gas cost when storing only content identifiers.

These results highlight the trade-offs between on-chain and off-chain storage. Storing files on the blockchain incurs higher costs and lower throughput, while integrating IPFS provides a more scalable and cost-efficient solution. For applications requiring large data storage, such as those combating counterfeit products, IPFS and blockchain integration offer an optimal balance of cost-effectiveness, scalability, and data integrity. The experimental results validate the proposed system's design, showing that IPFS integration significantly improves the scalability and efficiency of the blockchain-based solution for tracking product authenticity. The combined use of decentralized IPFS storage and the Ethereum blockchain ensures that the system can handle large-scale deployments while maintaining economic viability and data integrity.

VI. CONCLUSION

In this paper we have presented a distributed system to assure supply chain immutability and integrity by leveraging on blockchain and smart contracts for implementing basic operations of tracking products and IPFS for storing large volumes of data. Our approach offers a generic framework for storing data regarding composite products and defines a set of standard operations such as adding new products, updating products and searching products by displaying their history. A set of experiments is performed on a test-net with similar parameters to the public network, to evaluate the costs and the scalability of the system.

The original system, with full implementation on Ethereum blockchain, shows a throughput of 11 TPS for operation of adding product, and a cost of 2.58\$. To increase the scalability of the system and to lower its costs, the blockchain was integrated with IPFS, obtaining a reduction of costs of roughly 50% and almost doubling the throughput. Thus, for the add product operation, the cost would be 1.13\$ and the throughput of the system is increased to 26 TPS.

VII. REFERENCES

- [1] C. A. Burgos and N. Wajzman, Eds., Economic impact of counterfeiting in the clothing, cosmetics, and toy sectors in the EU. European Union Intellectual Property Office, 2024
- [2] Safety Gate: The EU Rapid Alert System for Dangerous Non-Food Products <https://ec.europa.eu/safety-gate/alerts/screen/search?resetSearch=true> (Last accessed 30-06-2024)
- [3] Pfizer Inc., 2019 Financial Report, Available online https://www.annualreports.com/HostedData/AnnualReportArchive/p/NYSE_PFE_2019.pdf
- [4] Lee, Heongu, and Changhak Yeon. 2021. "Blockchain-Based Traceability for Anti-Counterfeit in Cross-Border E-Commerce Transactions" *Sustainability* 13, no. 19: 11057. <https://doi.org/10.3390/su131911057>
- [5] Patidar, A.; Sharma, M.; Agrawal, R.; Sangwan, K.S. A Smart Contracts and Tokenization Enabled Permissioned Blockchain Framework for the Food Supply Chain. In *APMS 2021: Advances in Production Management Systems: Artificial Intelligence for Sustainable and Resilient Production Systems*; Springer: Cham, Switzerland, 2021
- [6] X. Yang, M. Li, H. Yu, M. Wang, D. Xu, and C. Sun, "A trusted blockchain-based traceability system for fruit and vegetable agricultural products," *IEEE Access*, vol. 9, pp. 36 282–36 293, 2021.
- [7] K. Biswas, V. Muthukkumarasamy, and W. L. Tan, "Blockchain based wine supply chain traceability system," in *Future Technologies Conference (FTC) 2017. The Science and Information Organization*, 2017, pp. 56–62.
- [8] Haq, Esuka, Blockchain Technology in Pharmaceutical Industry to Prevent Counterfeit Drugs, *International Journal of Computer Applications* (0975 – 8887) Volume 180 – No.25, March 2018
- [9] C. Antal, T. Cioara, M. Antal, and I. Anghel, "Blockchain platform for covid-19 vaccine supply management," *IEEE Open Journal of the Computer Society*, vol. 2, pp. 164–178, 2021.
- [10] A. Musamih, K. Salah, R. Jayaraman, J. Arshad, M. Debe, Y. Al-Hammadi, and S. Ellahham, "A blockchain-based approach for drug traceability in healthcare supply chain," *IEEE Access*, vol. 9, pp. 9728–9743, 2021.
- [11] S. Sadri, A. Shahzad, and K. Zhang, "Blockchain traceability in healthcare: Blood donation supply chain," in *2021 23rd International Conference on Advanced Communication Technology (ICACT)*, 2021, pp. 119–126.
- [12] I. A. Daniel, C. Pop, I. Anghel, M. Antal and T. Cioara, "A Blockchain based solution for Managing Transplant Waiting Lists and Medical Records," *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, Cluj-Napoca, Romania, 2020, pp. 505-510, doi: 10.1109/ICCP51029.2020.9266214.
- [13] Y. Zuo, "Tokenizing Renewable Energy Certificates (RECs)—A Blockchain Approach for REC Issuance and Trading," in *IEEE Access*, vol. 10, pp. 134477-134490, 2022, doi: 10.1109/ACCESS.2022.3230937.
- [14] T. Igarashi, T. Kazuhiko, Y. Kobayashi, H. Kuno, and E. Diehl, "Photrace: A blockchain-based traceability system for photographs on the internet," in *2021 IEEE International Conference on Blockchain (Blockchain)*, 2021, pp. 590–596.
- [15] Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. pp. 395. ISBN 0-201-63361-2
- [16] Ethers.js <https://docs.ethers.org/v5/> (Last accessed: 13-05-2024)
- [17] What is Ethereum? <https://aws.amazon.com/blockchain/what-is-ethereum/> (Last accessed: 28-06-2024)
- [18] MetaMask <https://docs.metamask.io/> (Last accessed: 14-05-2024)
- [19] Infura <https://www.infura.io/> (Last accessed: 14-05-2024)
- [20] Sepolia testnet <https://www.alchemy.com/overviews/sepolia-testnet> (Last accessed: 12-05-2024)
- [21] Goerli testnet <https://www.geeksforgoerli.com/what-is-goerli-testnet/> (Last accessed: 12-05-2024)
- [22] Hardhat <https://hardhat.org/> (Last accessed: 12-05-2024)
- [23] Composite Design Pattern <https://refactoring.guru/design-patterns/composite> (Last accessed: 30-06-2024)
- [24] IPFS – InterPlanetary File System <https://ipfs.tech/> (Last accessed 30-06-2024)