National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Creation of a user data market system through the Cardano network

Diploma Thesis

of

MARKOS GIRGIS

Supervisor: Nectarios Koziris
Professor NTUA

Athens, November 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Creation of a user data market system through the Cardano network

Diploma Thesis

of

MARKOS GIRGIS

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the examination committee on 21st November 2023.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Nectarios Koziris	Aggelos Kiayias	Aris Pagourtzis
Professor NTUA	Professor University of Edinburgh	Professor NTUA

Athens, November 2023

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Copyright © – All rights reserved.
Markos Girgis, 2023.

The copying, storage and distribution of this diploma thesis, exall or part of it, is prohibited for commercial purposes. Reprinting, storage and distribution for non - profit, educational or of a research nature is allowed, provided that the source is indicated and that this message is retained.

The content of this thesis does not necessarily reflect the views of the Department, the Supervisor, or the committee that approved it.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

.....
Markos Girgis

21st November 2023

Abstract

In an era where data is often described as the new oil, the control and monetization of personal data have largely been in the hands of centralized entities. This thesis presents a decentralized data marketplace built on the Cardano blockchain, leveraging smart contracts to automate the process of data exchange between buyers and sellers. Utilizing Haskell and Plutus for smart contract development, and a browser extension for data collection, the system aims to return control of personal data to the individual.

Utilizing Cardano's Extended UTXO (EUTxO) model and smart contracts, the marketplace enables users to tokenize their data, thereby converting it into tradable assets. The system incorporates a browser extension for data collection, IPFS for decentralized data storage, and cryptographic techniques for secure data transmission and identity verification. Two primary workflows, "Ask" and "Bid", are elaborated to demonstrate the data sale and purchase mechanisms.

This diploma thesis aims to indicate the potential of blockchain technology and smart contracts in automating complex processes, thereby offering a versatile solution to the universal problem of data ownership and monetization. This work serves as a stepping stone for further research in leveraging blockchain technology for practical, secure, and transparent data management solutions.

Keywords

Blockchain, Smart Contracts, Cardano, Decentrilized Data Marketplace, Data Ownership, Haskell, Plutus, IPFS

to my parents

Acknowledgements

I would like to thank first of all Prof. Koziris for the supervision of this thesis and for the opportunity he gave me to conduct it in the Computing System Laboratory. Special thanks are due to Dr. Doka and Mr. Katsigiannis for their guidance and the excellent cooperation we had. I am also grateful for the funding provided by IOG, which has supported my research endeavors. Finally I would like to thank my parents for the guidance and moral support that they have given me over the years.

Athens, November 2023

Markos Girgis

Table of Contents

List of Figures

List of Images

List of Tables

Preface

In the digital age, data has become one of the most valuable commodities. Yet, the individuals who generate this data often have little control over how it is used, monetized, or even secured. This imbalance of power between data generators and data controllers has led to countless of ethical and privacy concerns that are becoming increasingly difficult to ignore. At the same time, the rise of blockchain technology has shown promise in removing central authorities and empowering individuals, offering a potential solution to this pressing issue.

My journey into the world of blockchain technology was furtherly motivated by a fascination with the capabilities of smart contracts to automate and secure complex processes. While data privacy is an important aspect, the primary aim of this thesis is to showcase how smart contracts can be practically applied to solve universal problems—in this case, the issue of data ownership and monetization.

As you go through the pages that follow, you will find a comprehensive exploration of a decentralized data marketplace built on the Cardano blockchain. This marketplace leverages smart contracts to give individuals greater control over their personal data, allowing them to monetize it on their terms, while also automating the data transaction process and enhancing security. Although the implementation is specific to the Cardano ecosystem, the principles and methodologies can be adapted to other blockchain platforms, offering a versatile solution to a universal problem.

I hope this work serves as an inspiration for further research and development in leveraging smart contracts for practical solutions. Whether you're a student, a researcher, or simply someone intrigued by the transformative potential of smart contracts to automate and secure a wide range of applications, I believe this thesis has valuable insights to offer you.

Thank you for taking the time to engage with this work. I look forward to the discussions and developments that it may spark.

Chapter 1

Introduction

In the modern era, data has become one of the most valuable commodities. It drives decision-making in various sectors, from healthcare and finance to marketing and public policy. However, the current models of data collection and monetization often leave individual users with little control over their personal information. This centralized approach has led to growing concerns about data privacy, security, and ownership. The rise of blockchain technology and smart contracts offers a transformative solution to these challenges, enabling a decentralized, transparent, and secure framework for data management.

While existing systems offer some level of data protection, they often rely on centralized intermediaries, making them susceptible to security risks and limiting user control. Moreover, users rarely receive tangible benefits for contributing their data, creating an imbalance in the data economy.

The primary aim of this thesis is to explore the potential of blockchain technology, specifically through the use of smart contracts on the Cardano network, to create a decentralized data marketplace. This marketplace aims to:

1. Empower users by giving them control over their data through smart contract automation.
2. Provide a transparent and equitable system for data exchange, eliminating the need for centralized intermediaries.
3. Ensure security by leveraging the cryptographic features of blockchain technology.

Research Questions

1. How can blockchain technology be leveraged to create a decentralized data marketplace?
2. What are the technical challenges and solutions in implementing such a marketplace?
3. How can smart contracts ensure data privacy and security?

This thesis focuses on the design and implementation of a decentralized data marketplace using Cardano's smart contract platform, Plutus. It covers the technical aspects,

including front-end and back-end development, smart contract implementation, and data storage solutions. Additionally, it explores the ethical considerations surrounding data privacy and ownership in a blockchain-enabled environment.

Thesis Outline

1. Introduction: Provides an overview of the project, including the problem statement, objectives, research questions, and scope of the study.
2. Background: Explores the theoretical foundations of the project, including blockchain technology, smart contracts, and decentralized applications.
3. Methodology and Implementation: Details the system architecture and development process.
4. Results and Discussion: Analyzes the technical achievements, challenges, and future directions.
5. Ethical Considerations: Explores the ethical implications of the project.
6. Epilogue: Reflects on the broader impact and potential for future work.

By addressing the challenges and opportunities in creating a decentralized data marketplace, this thesis aims to contribute to the ongoing discourse on data privacy, security, and ownership. It serves as a stepping stone towards a future where blockchain technology and smart contracts empower individuals with greater control over their personal data, thereby fostering a more transparent and equitable data economy.

Part I

The Theoretical Background

Chapter 2

Literature Review

2.1 Blockchain Technology

Blockchain has emerged as a groundbreaking technology, in both the academic and the business world , fundamentally altering the way we perceive digital transactions and data storage. It gained initial prominence through its first application in cryptocurrencies like Bitcoin [1]. The technology is characterized by its decentralization, transparency and robust security mechanisms [2].

However Blockchain technology has found applications far beyond cryptocurrencies, extending into sectors like healthcare, supply chain management and governance [3]. In healthcare for example blockchain is employed to ensure data privacy and security, especially when it comes to sensitive patient data [3]. The decentralized storage mechanism of blockchain ensures that data is not only secure but also easily verifiable, which adds an extra layer of trust and accountability.

Blockchain, at its core functions as a ledger whose purpose is to record transactions across a network of distributed computing systems. This unique architecture provides security, transparency and data integrity. To achieve this transactions are grouped into blocks, then connected in a sequential manner to create a chain like structure called "blockchain." Unlike ordinary systems, blockchain doesn't rely on a central authority for control, making it more democratic. The authenticity of transactions is verified by network nodes using cryptographic techniques and consensus algorithms which strengthens the network against changes and ensures its reliability.

2.1.1 The UTXO model

Bitcoin, the crown jewel of cryptocurrencies, operates on a ledger model that is fundamentally rooted in the concept of Unspent Transaction Outputs (UTxOs) [4]. In this model, individual transactions are not mere linear events; they are complex entities consisting of both inputs and outputs. These outputs serve as value placeholders, representing specific cryptocurrency amounts that stand ready to be utilized in future transactions.

It is important to note that each output is destined to be linked to precisely one input in a subsequent transaction. It's a meticulously structured system, with no cycles allowed, and no double spendings. This ensures that the transactions form a Directed Acyclic Graph (DAG), a graph of financial interactions where each transaction—characterized by

'm' inputs and 'n' outputs—manifests as a unique node, complete with its own set of inbound and outbound edges. An output can then serve as a potential input for subsequent transactions, thus contributing to the continuous construction of the blockchain.

The model also enforces a law of conservation, akin to the physical laws that govern our universe. The total value absorbed by the inputs of a transaction must be in perfect equilibrium with the total value emitted by its outputs. In essence, value is neither created nor destroyed; it is meticulously conserved.

An important detail to note is: each output is a one-time-use entity. Once an output is used as an input in a subsequent transaction, it transitions from being "unspent" to "spent." This means it can no longer be reused as an input for any future transactions. This one-time-use nature of outputs is what gives them the label "Unspent Transaction Outputs" until they are actually spent. It's a built-in security feature that prevents double-spending and ensures the integrity of the transaction history.

2.2 Smart Contracts

2.2.1 The Extended UTXO Model

Cardano's ledger model is the EUTxO [5].The Extended UTxO model (EUTxO) is an extension of Bitcoin's UTXO model that supports a more expressive form of validation scripts. In this extended model, an output can be spent, used as input of a subsequent transaction, only if it satisfies a function v. This function is known as the output's validator. Here comes the concept of the redeemer. A transaction proves its eligibility to spend an output by providing a redeemer value ρ, such that $v(\rho)$ = true.

Let's examine the simplest possible case, where a wallet owner tries to spend one of their own UTXOs. In a basic UTXO model, one can conceptualize the redeemer being the cryptographic hash of the spending transaction signed by that wallet's private key and the validator function as a fixed script that verifies, if this hash is signed by the owner of the transaction's inputs.

On the other hand, in the Extended UTxO (EUTxO) model, the concept of a redeemer is not confined to a specific value; it can essentially be any value, chosen and dispatched by the transaction aiming to spend the UTXO. Moreover, the validator function v is not set in stone; it can be substituted with any smart contract logic that suits the application's needs.

To capture and maintain the state of the machine, the EUTxO model elevates the traditional UTxO output from a mere pair of a validator v and a cryptocurrency value to a more intricate triple—$(v, \text{value}, \delta)$. Here, δ is a datum that holds contract-specific data, adding another layer of complexity and utility to the model.

But that's not all! In Cardano's implementation, smart contracts are also aware of the entire context of the spending transaction that is attempting to consume the UTXO. This comprehensive access empowers validators to enforce contract continuity seamlessly. Multiple validators can be coordinated to create complex systems. To sum it up, for an input with a redeemer ρ that is part of the transaction tx, the system verifies its

entitlement to spend an output $(v, \text{value}, \delta)$ by ensuring that

$$v(\text{value}, \delta, \rho, \text{tx}) = \text{true}.$$

In an attempt to summarize the essence of the Extended UTxO (EUTxO) model and its role in Cardano's smart contracts, let's synopsize its core components:

- Datum: This is not just any piece of data; it's a payload carried within a transaction output. Set in stone by the transaction that births the output, the datum is immutable. When an output is about to transition into a "spent" state, this datum is passed along to the validator function, which then decides whether to greenlight its spending. This mechanism provides each locked output with a certain degree of statefulness.
- Context: The validator isn't operating in a vacuum; it receives rich information concerning the spending transaction. This includes all of the transaction's inputs and, crucially, access to all of its outputs. This opens the door to interesting scenarios where multiple validators can work in coordination to orchestrate complex logical operations.
- Redeemer: This is the wildcard value passed to the validator by the spending transaction. It's a versatile tool, especially when multiple actors wish to invoke the same validator. The validator can apply distinct rules to each actor, or even allow a single actor to execute varied operations.

2.2.2 Cardano and Plutus

Cardano employs Plutus as its native programming language for smart contracts, a powerful tool based on Haskell, which makes good use of Haskell's strong type system and functional programming paradigms. In practice, smart contracts are developed in Haskell, which is then compiled into Plutus Core using the Plutus Tx GHC plug-in. Unlike Etherium's Solidity, Plutus places great priority into security. This section dives into Plutus' strengths and some of its advantages over Solidity, as highlighted in the paper "UTxO- vs account-based smart contract blockchain programming paradigms" by Brünjes, Gabbay, and others (Brünjes et al., 2020) [6].

Enhanced Programmability Plutus places a higher burden on programmers but compensates with robust mathematical properties that enhance programmability, like Monads [6]. This mathematical emphasis provides a strong foundation for developing more complex and secure smart contracts.

User control - Determinism One of the standout features of Plutus is the level of control it offers to users. In Plutus, the consumer of a contract can create a transaction and determine its inputs and outputs in advance. This allows users to guard against potential errors or attacks independently of the contract's designer [6]. Whenever a transaction is

accepted, it will have predictable effects on the state of the Ledger. By making sure a script will always terminate(see Halting Problem [7]) and that it will return the same result if applied the same arguments, like a pure function, deterministic script evaluation is provided. Consumers can run locally their potential transaction, and predict how much the fees will cost, if the transaction will be accepted and the impact it will have on the Ledger. Thus, users acting in good faith will have no unexpected side effects and will never lose any collateral by accident.

Mission-Critical Impact Mission-Critical Impact The paper points out that Ethereum's can be "arguably buggy", a phenomenon stemming from the underlying programming paradigm of Solidity. Plutus avoids such risks, making it a more reliable for mission-critical applications [6].

In addition to its mathematical rigor and user-centered design, Plutus also inherits Haskell's focus on immutability. This feature is not just a theoretical nicety, but a practical feature that enhances the security and robustness of smart contracts. Immutability removes a wide range of bugs related to mutable state, making it easier to reason about the behavior or smart contracts while developing them, and ensuring that once deployed the contract's rules will behave as expected. This aligns well with blockchain technology's immutable nature, a key feature responsible for its security and trustiness.

2.2.3 Validators and Minting Policies

So far we have discussed Validator smart contracts, and we mentioned that whenever a UTxO locked by a validator smart contract address is attempted to be spent, the corresponding validator smart contract will run, with 3 inputs, the Datum(from UTxO), the Redeemer(from input) and the Transaction Context.

However a smart contract apart from having a Validator purpose, it can also have a Minting Policy purpose. These smart contracts are called Minting Policies, and they give the ability to mint native Cardano tokens, which can serve multiple purposes in the blockchain ecosystem.

Tokens can be used to represent specific roles for users interacting with smart contracts [8]. For instance, a token could represent the role of a creditor in an interest-based contract. Since a token can have a unique identifier, smart contract datums can point to them, creating complex role based conditions, without requiring a hard-coded list of assets inside the contract itself. Since tokens are themselves resources on the ledger, they can be traded, effectively allowing for the swapping of roles [8]. For instance, a buyer could acquire a seller token, as will be demonstrated in the upcoming implementation. Tokens can ensure that all participants in an agreement, such as initial coin offering, have been involved. By issuing participation tokens, the right to participate becomes a tradable asset, ensuring that no participant can be unfairly omitted [8].

Concerning their execution context, Minting Policies run whenever someone tries to mint or burn a token(Currency symbol) they have defined. They receive 2 inputs, the Redeemer and the Transaction Context. They are not validating the unlocking of some

UTxO, so there's no Datum, in contrast to Validator scripts.

2.2.4 Parameterized scripts

So far we have discussed smart contract validators, which have a validator address and lock UTxOs sent at their address.Instead of defining one script, with a single script address, with all UTxOs sitting at the same address, we can define a family of scripts that are parameterized by a given parameter [9]. These are called parameterized scripts. A parameterized script, once passed actual concrete parameter values, can obtain its address like simple un-parameterized validator scripts do. This ability, raises the question, whether we should sometimes put a piece of state we want into the datum or into a script's params.

When we add the data in the datum, and stick with non-parameterized script, we have 1 script address. So all UTxOs and Datums for a non-parameterized script sit at the same script address, meaning they are easily discoverable.

On the other end, if we use a parameterized script, for each different choice of parameters, we get a different script, with a completely different hash and therefore a completely different address. As a result they are much harder to find for someone who would want to discover them. They would need to know which params to look for in order to compute the script's address. So overall choosing between the two can depend on the following factors:

1. Type of Data: If the data you're dealing with is dynamic and expected to change frequently during the contract's operation, then it may be more suitable to store this data in the Datum. The Datum is designed to hold the state of the contract at any given time, and so it is well-suited for frequently changing data. On the other hand, if the data is more static and set when the contract is first deployed(like a specific ratio for a decentralized exchange), then it may be better to use a parameterized script.

2. Contract Flexibility: Parameterized contracts offer greater flexibility. They can behave differently based on the parameters they are instantiated with. If you want to reuse the contract logic across different instances with slightly different behaviors, then a parameterized contract can be a good choice. However, if your logic is tightly coupled with the state of the contract and the state changes frequently, using the Datum to store state might be a better approach.

3. Size and Complexity: The complexity and size of your data can also influence this decision. If your data is large or complex, storing it in the Datum might be more manageable and efficient.

In general, these decisions will often be made based on the specific requirements of the smart contract, and there's no one-size-fits-all answer. Understanding the contract's domain, as well the differences and trade-offs of each option is vital to guide your decision.

Minting Policies make great use of parameterized scripts. By parameterizing a minting policy with the id of a user's specific UTxO, and then demanding that any transaction using that minting policy must consume as input that UTxO, truly unique tokens can be minted. A UTxO can only be spent once, based on the UTXO model, so this ensures that only the owner of the UTxO parameter will be able to mint the token, and do that only once. This approach ingeniously enables the creation of authentic non-fungible tokens (NFTs) within the Cardano ecosystem.

2.2.5 Off-chain vs On-chain code

Plutus Core code operates in two distinct realms: the on-chain and off-chain environments, each with its own set of rules and objectives.

When we talk about on-chain code, we're referring to the computational logic that validates transactions directly on the blockchain. This code runs every time a transaction is proposed, assuring its validity. If the transaction passes, it's highly likely to be incorporated on the blockchain. But this computational validation isn't free, transaction fees are levied to deter malicious actors from flooding the network and unnecessarily occupying the blockchain nodes [10], thereby safeguarding the blockchain's operational integrity.

On the flip side, off-chain code has a different mission. It serves as the architect of transactions, crafting them for submission to the blockchain. Imagine you're engrossed in an online auction hosted on Cardano. To place a bid, you'll need to dispatch specific data to the auction's smart contract—like an ID and your Ada offer. Here, off-chain code takes the reins, constructing the transaction, running preliminary validations to avoid fee wastage, and calculating the transaction fees before sending it off into the blockchain.

Firstly, consider a user locking a UTXO under a specific validator address. When the moment arrives to spend this UTXO, the spending validator is invoked. But here's the catch: this validation process initially occurs off-chain. Why? To mitigate the computational burden on the blockchain nodes. This off-chain validation serves as an initial filter, ensuring that only transactions with a high likelihood of success proceed to the on-chain validation stage. It's a resource-efficient mechanism that prevents the network from being overwhelmed with failing transactions.

In the event that a transaction bypasses the off-chain checks—perhaps due to a rogue actor attempting to abuse the system—they won't be able to bypass the on-chain validations. Each transaction involving smart contract validation is required to include a collateral UTxO [11]. Should the transaction fail the on-chain validation, this collateral UTxO is forfeited. It's a financial safeguard that adds another layer of security, ensuring that malicious actors think twice before attempting to flood the network with invalid transactions.

Smart Contract Deployement When interacting with smart contracts, there are two options: either embed the entire smart contract within each transaction or deploy it once and reference it in subsequent transactions, using a reference input UTxO. The latter approach, akin to a software design pattern, minimizes redundancy by storing

the validator script once on the blockchain and then referencing it as needed. While this demands an initial cost for deployment, it yields long-term savings in computational resources and storage, making it the preferred strategy for frequently used contracts. On the other hand, Minting Polcies that are meant to be run only once can be embedded in the transaction that mints the token, and then discarded, since they are not needed anymore.

In summary, the Extended UTxO model offers a robust framework for transaction validation, smart contract deployment, and token minting, optimizing for both computational efficiency and storage. It's a paradigm that aligns well with the principles of scalable, secure, and efficient blockchain systems.

2.3 Decentralized Applications

What are dApps? Decentralized applications, commonly known as dApps, are a groundbreaking form of software systems empowered by blockchain technology. Unlike traditional applications that run on centralized servers, dApps operate on a peer-to-peer (P2P) network. This decentralization is not merely a technical shift, it's a way that redefines how applications are structured and function. The immutability feature of blockchain is fundamental to these applications. This immutability ensures that once data is recorded, it cannot be altered without altering all subsequent blocks, thereby providing a robust layer of security.

While the initial killer application of blockchain was, undoubtedly, cryptocurrencies, the true potential of blockchain technology lies far beyond just digital coins. Decentralized applications offer a more secure, transparent, and open-source environment for users. They serve multiple purposes, from financial transactions and smart contracts to decentralized autonomous organizations (DAOs) and beyond.

One should note that decentrilized applications also come with their own challenges and vulnerabilities. One such issue includes the Byzantine Generals' Problem, a data synchronization issue in distributed systems [12], when tryint to achieve consensus among potentially unreliable or malicious network nodes. It should be noted, however, that most blockchain protocols implement consensus algorithms designed to mitigate this very issue.

2.4 IPFS Protocol

As more and more dApps immerse, the need for off-chain data storage is a recurring theme. While blockchain excels in ensuring data integrity through its immutability principle and establishing trust, it is not a suitable place for storing large volumes of data. This is where the InterPlanetary File System (IPFS) [13] comes into play, offering a robust solution that aligns well with the decentralized nature of blockchain.

Conceived by Juan Benet, IPFS is a peer-to-peer (P2P) distributed file system that aims to connect all computing devices under a unified, versioned file system. Unlike the familiar HTTP protocol, where a domain name essentially maps to a specific IP address

hosting the desired content, making it location centric, IPFS adopts a content centric approach. In IPFS users search for a unique content identifier, commonly abbreviated as CID, which essentially is a hash of the content in question. Rather than pointing to a specific address, the content is decentralized and distributed across the network's peers. Each network member maintains a distributed hash table(DHT), that lists the accessible locations for each CID. These hyperlinks (CIDs) and the data they point to form a Merkle DAG, a data structure that allows for efficient storage and retrieval of data in a decentralized network [13].

Why is IPFS used in dApps? IPFS (InterPlanetary File System) is more than just a system for distributing files across multiple computers, it combines several advanced features to create a robust and versatile platform for decentralized data storage and retrieval. Its components ensure that IPFS is resilient to single points of failure and eliminate the need for nodes to trust each other. It's a robust system that has already found applications in various domains, including blockchain technology [14].

Chapter 3

Related Work

3.1 Book.io: Tokenizing Digital Books for True Ownership

Book.io presents a compelling case study in the application of blockchain technology to digital asset ownership, specifically in the realm of digital books (eBooks and Audiobooks) [15]. Unlike traditional digital book platforms, where consumers purchase a "license to access content" rather than owning the actual content, Book.io introduces the concept of Decentralized Encrypted Assets (DEAs). These DEAs grant genuine ownership of digital books to consumers, allowing them to sell, lend, or give away their digital books.

Their platform employs a two-tiered technological architecture:

- The Decetralized Encrypted Asset (DEA), which represents the actual encrypted digital book.
- The $BOOK Token, serving as the ecosystem's utility and loyalty token.

The DEAs are a significant advancement over traditional Non-Fungible Tokens (NFTs). Unlike basic NFTs that merely link to an image or file, DEAs in Book.io are fully encrypted and stored in decentralized storage, ensuring that only the owner can access the content. This feature aligns closely with the data ownership and encryption aspects of our data sale marketplace, although applied in a different domain.

Furthermore, Book.io introduces a native $BOOK token to incentivize reading, offering a unique perspective on how tokens can drive user engagement and add value to digital assets. This token-based incentive mechanism could offer insights into enhancing user participation in data sale marketplaces.

In summary, Book.io serves as an instructive example of how blockchain technology can revolutionize digital asset ownership and user engagement, and provide immutable and trustless record-keeping. Its innovative use of DEAs and token-based incentives provides valuable lessons for the development and refinement of blockchain-based data sale marketplaces.

3.2 JPG Store

JPG Store emerges as a pioneering marketplace for Non-Fungible Tokens (NFTs) on the Cardano blockchain, introducing innovative features that gamify user interactions

and transactions. The platform is an ecosystem covering a wide range of digital artworks and creations, that rewards active participation through Experience Points (XP) and its native $JPG token.

The $JPG token serves multiple purposes within the ecosystem. It acts as a reward for loyal users and offers various benefits, such as reduced trading fees and priority minting. This multi-utility token could provide insights into how a similar token could be implemented in a data sale marketplace for incentivizing data providers and consumers.

JPG Store has undergone multiple smart contract upgrades to enhance user experience and security. The most recent upgrade focuses on performance, allowing for bulk purchases of up to 52 assets in a single transaction. This is particularly relevant to our data sale marketplace, where efficient and secure transactions are crucial.

JPG Store aslo employs a dual validator system for handling "asks" and "bids," terms borrowed from traditional finance. The dual validator system in JPG Store is designed to optimize transaction efficiency and security. This technology could offer valuable insights for our marketplace, where similar challenges around transaction speed and security exist.

In summary, its innovative features and mechanisms provide valuable insights that could be applied to enhance user engagement and transaction efficiency in blockchain-based data sale marketplaces.

Part II

Methodology and Implementation

Chapter 4

Methodology and System Architecture

The architecture of this project is an orchestration of several components: a browser extension, a Next.js dApp with both front-end and back-end capabilities, and three Plutus smart contracts—two for validation and another for minting policies. Additionally, the InterPlanetary File System (IPFS) is employed for data storage, and a key-value storage system is utilized by the back-end.

4.1 Overview of the dApp

Broadly speaking, the system accommodates two types of actors: Sellers and Buyers. A Seller is an individual who aims to monetize their browsing and personal data by selling it. On the other hand, a Buyer is an entity or individual interested in purchasing such data for analytics or other purposes.

In the architecture of this decentralized marketplace, there are two primary transactional mechanisms that facilitate the exchange between a seller and a buyer: the "Ask" and the "Bid" flows.

1. Ask Flow: In this model, the seller takes the initiative by locking a specific token in a smart contract. The token is essentially "listed" with a predetermined price tag inside its Datum. This sets the stage for buyers to meet this price in order to unlock and acquire the token. The DataListing validator smart contract ensures that the token is securely held until the asking price is met, at which point the token is transferred to the buyer and the agreed-upon sum is sent to the seller.

2. Bid Flow: Contrary to the Ask flow, the Bid model is buyer-centric. Here, buyers can place bids on tokens they are interested in. The bid consists of a certain amount of ADA that the buyer is willing to pay for the token. Sellers can browse these bids made on their tokens and choose to accept any that meet their valuation of the token. Upon acceptance, the smart contract facilitates the immediate exchange of the token and the bid amount between the seller and the buyer.

Both flows offer unique advantages and appeal to different trading preferences, thereby creating a versatile and dynamic marketplace.

Components of the System:

1. Browser Extension: Used by the Seller, this extension captures the user's browsing data. The user inputs their wallet address and selects the duration for which they want to capture their browsing history. Currently, the data captured is limited to direct visits—URLs explicitly entered by the user. However, the Chrome API offers the flexibility to capture a richer set of data in the future.

2. Smart Contracts: Developed in Plutus (a Haskell-based language), the system employs three smart contracts. The first is a minting policy for generating unique "DataToken" tokens. The second and third are validator smart contracts, one for handling Asks(DataListing) and the other for Bids. These contracts ensure secure and fair transactions between Sellers and Buyers and also include a "Cancel" mechanism for both parties.

3. Back-end Server: Written in TypeScript(NodeJS), the back-end server of the marketplace is built using Next.js's API. It encrypts the data received from the extension and stores it on the IPFS network. It also handles token metadata, which is crucial for data retrieval and decryption for the Buyer.

4. Also written in TypeScript, the User Interface is developed using Next.js and styled with Tailwind CSS. It serves as the hub for all off-chain code, handling transactions, locating necessary UTXOs and tokens, and submitting transactions to the network. The lucid-cardano library is used to facilitate these operations. The interface is divided into 2 primary views, one designed for the Seller and the other for the Buyer.

By integrating these components, the system offers a comprehensive solution for the secure and efficient buying and selling of user data.

4.2 Browser Extension

A browser extension is essentially written using the same web technologies used to create a web application such as HTML, CSS, and JavaScript. However, what sets it apart is its ability to have access into specialized Browser APIs, like Chrome's API for Google Chrome extension.

Additionally, the Content Security Policy (CSP) in browser extensions adds an extra layer of complexity compared to traditional web development [16]. For instance, the default CSP restricts extensions to only load local scripts and objects, disallowing inline JavaScript and the evaluation of strings as executable code. This means that common JavaScript functionalities like eval() are off-limits, and the same goes for any libraries that rely on such features. These limitations are designed to enhance security but can pose challenges during development, requiring a more cautious approach to script inclusion and execution.

The most key components of a browser extension [17, 18] are:

1. The manifest: The cornerstone of any extension is its manifest file, conveniently named manifest.json. Situated in the root directory, this file serves as the blueprint

for the extension, detailing its metadata, permissions, and the files it needs to run both in the background and on web pages. It is basically a configuration file defining an extension's architecture and components.

2. Service Worker: Acting as the extension's event manager, the service worker listens for various browser events, such as tab creation or bookmark removal. While it can leverage all Chrome APIs, it can't directly manipulate web page content; that's the job of content scripts.

3. Content scripts: These scripts execute JavaScript within the context of any visited web page, allowing them to read and modify the DOM. Although they can only use a subset of Chrome APIs, they can still indirectly access the full range by exchanging messages with the service worker.

4. Popup and Other HTML Pages: Extensions can include a variety of HTML files, like popups or options pages. These serve as the primary user interface for interacting with the extension and can access Chrome APIs.

Manifest V3 Manifest V3 serves as the most up-to-date framework for Chrome extensions, offering improvements in security, privacy, and performance. It also enables the use of modern web technologies like service workers and promises. The manifest file is pivotal as it outlines the extension's capabilities and required permissions, which are presented to the user upon installation. Extensions operate in a sandboxed environment, restricting access to only the necessary resources.

The popup component acts as the primary user interface, appearing when the extension icon is clicked. It's limited in that it can't engage with the web page's DOM or interact with other extensions.

On the other hand, the content script has the ability to manipulate a web page's content but lacks access to Chrome's API.

Service workers function as background scripts, orchestrating various extension activities and responding to browser events. While they can't interact with the DOM, they serve as the extension's event manager, listening for occurrences like new tab creation, bookmark additions, or extension icon clicks.

Each component within the extension comes with its own set of strengths and constraints. To fully leverage the extension's capabilities and functionalities, these components must engage in coordinated interactions with one another. This interaction is orchestrated through message-passing, echoing the principles of an event-driven architecture. The popup, for instance, can capture an event triggered by the user and broadcast a corresponding message. This message can then be intercepted by a service worker for data storage purposes or by a content script to modify the webpage's content.

4.3 NextJS dApp

As the world of decentralized applications (dApps) continues to evolve, the need for robust and scalable frameworks becomes increasingly evident. Next.js, a leading server-side rendering (SSR) framework for React that offers a compelling solution for dApp development.

Unified Codebase One of the standout features of Next.js is its ability to house both frontend and backend logic within the same codebase. This co-location simplifies development workflows and enhances code maintainability. It allows developers to write API routes and server-side functions alongside their React components, enhancing the development process and reducing context switching.

Frontend Technologies The frontend of the dApp is built using React, a popular library for building user interfaces. It is styled using Tailwind CSS, a utility-first CSS framework, that provides out of the box css classes ready to be used. State management and context are handled using React's built-in Context API, ensuring efficient data flow and state handling across components.

Backend API Routes The backend is structured as a set of API routes, each serving a specific function. These routes have the capability to interact with various services, including IPFS for decentralized storage and a key-value storage system for token metadata.

Authorization Mechanisms Security is a paramount concern, especially in the context of dApps. The backend API routes employ various authorization mechanisms to validate requests coming from the client side. This ensures that only authenticated users can perform certain actions, adding an extra layer of security to the system.

Server-Side Rendering and React Next.js takes the power of React a step further by offering server-side rendering out of the box. This feature enhances the performance and SEO of web applications, a crucial aspect often overlooked in the realm of dApps. With SSR, the initial HTML content is generated on the server, reducing the time to first paint and improving the user experience.

TypeScript Support TypeScript, a superset of JavaScript, adds static types to the language, making it easier to catch errors during development rather than at runtime. The integration of TypeScript in a Next.js project is seamless, requiring just a simple configuration file to get started.

By adopting this architecture, the Next.js dApp achieves a harmonious blend of scalability, performance, and security, making it well-suited for modern decentralized applications.

4.4 Plutus Smart Contracts

The smart contract architecture is designed with precision and purpose, encapsulating the core logic and interactions taking place in the marketplace.

Data Token Minting Policy This is a parameterized minting policy smart contract, which as type of smart contract suggests, is responsible for minting the "DataTokens", used to represent and later access a user's stored browsing history data. Once parameterized with one of the Seller's available UTxOs , the contract generates a specific minting policy address, granting exclusive minting rights to that Seller. Importantly, this is a one-time operation; after the initial minting, the UTXO becomes spent, rendering it unusable for future minting.

DataListing Smart Contract The DataListing validator smart contract implements the 'Ask' flow in the marketplace's bid-ask model. It defines a Datum structure that encapsulates the seller's public key hash (PubKeyHash) and the asking price in lovelaces. The contract also recognizes two redeemer values: 'Purchase' and 'Redeem.' The 'Purchase' redeemer is meant to be used by interested potential Buyers, while 'Redeem' allows Sellers to withdraw their DataToken listing. The Datum's PubKeyHash and price fields are instrumental in validating the transaction's integrity, ensuring the buyer has met the payment requirements.

Bid Smart Contract The Bid validator smart contract is responsible for the 'Bid' flow and is parameterized by a token's unique asset class. This design choice enhances discoverability for both buyers and sellers by generating distinct smart contract addresses for each token. It sidesteps the logistical complications that would arise from a monolithic contract handling all possible bids for all tokens. The Datum in this contract contains the buyer's PubKeyHash, which is essential for verifying that the seller transfers the token when claiming the bid. Similar to the DataListing contract, the Bid contract also has two redeemers: 'Sell' and 'Redeem'. The 'Sell' redeemer is exclusive to the token owner (the Seller), while 'Redeem' is reserved for the buyer.

Security Measures Each validator smart contract incorporates robust security measures, including signature verification, to ensure that transactions are authorized by the appropriate parties. This adds the necessary layer of trust and reliability to the marketplace's decentralized architecture.

4.5 Data Flow Diagrams

Data flow diagrams, became popular in the 70s, because they provide a straightforward way to visualize the flow of data through a system. On a DFD, there are four main elements:

- External Entity: Represented by a rectangle, this element refers to the source or destination of data. In this system's architecture, the external entities are the Seller and the Buyer.

- Process: Represented by a circle, this element refers to an activity that transforms incoming data flow(s) into outgoing data flow(s). Usually processes start from the top left of the DFD and finish on the bottom right.

- Data Store: Represented by two parallel lines, this element refers to data that is stored within the system. In this system architecture, the data store is the IPFS network and the server's key-value(KV) storage.

- Data flow: Represented by an arrow, this element refers to the movement of data between external entities, processes, and data stores. Labels describe the type of data flowing. However, in many systems, especially those involving human interactions like user interfaces, the "data" being transferred can also be an action or an event. This is especially true for high-level DFDs (Level 0 or Level 1) where processes can be more abstract.

Seller interaction with the Browser Extension In Figure 4.1 we represent a Seller's interaction with the Browser Extension and the Server, in order to store their browsing history data on IPFS. The Seller's wallet address is used as a key to store the CID of the data on the Server's KV(Key Value) Storage.

For both flows, the Seller then enters the dApp and creates a DataToken, which is minted using the DataToken Minting Policy.

Ask Flow In the Ask Flow (Figure 4.2), the Seller then lists the DataToken for sale, by locking it under the DataListing Smart Contract. The Buyer can browse through locked tokens and purchase any that they are interested in. Upon purchase, the smart contract facilitates the immediate exchange of the token and the asking price between the seller and the buyer.

Bid Flow In the Bid Flow(Figure 4.3), the Buyer can make bids on tokens minted by Sellers, by locking their ADA offer under the Bid Smart Contract. The Seller can browse the bids made on their DataToken and accept any that meets their valuation of the token. Upon acceptance, the smart contract facilitates the immediate exchange of the token and the bid amount between the seller and the buyer. An important difference here is that the token is never locked under the Smart Contract, it is contstantly in the Seller's possession, until the moment of the exchange.

One subtle detail not depicted in the diagrams, for the sake of not overcomplicating them, is the creation of the off-chain metadata for each token upon the Token Minting, on the server's KV Storage. This metadata is crucial for the Buyer to be able to retrieve and decrypt the data stored on IPFS, on the post-purchase phase.

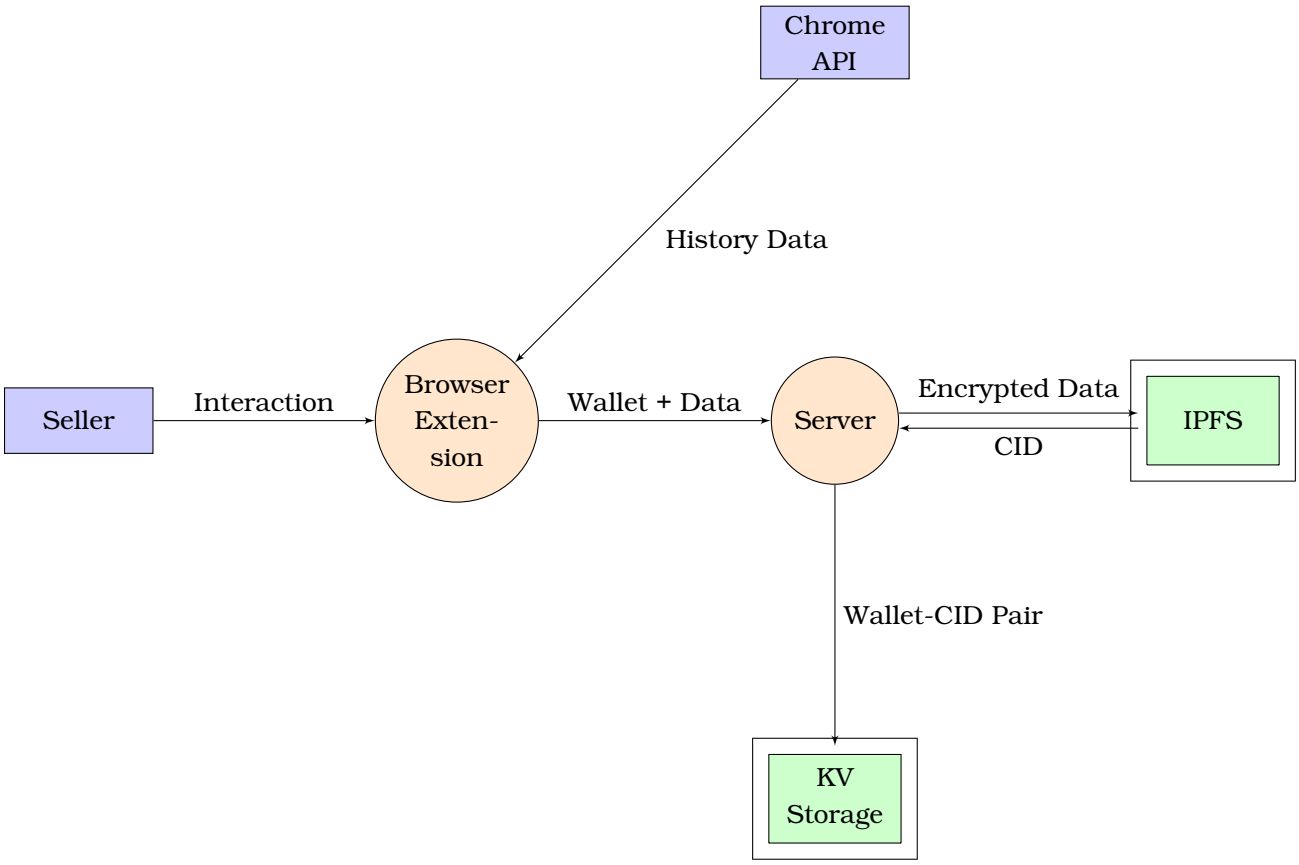

Figure 4.1. *Data flow diagram of a Seller's interaction with the Browser Extension.*

Post-Purchase At both flows, the Buyer having purchased the token, can proceed to download the data associated with this token.They need to send a digital signature to the backend server, in order to verify their identity. The server then fetches the encrypted data from IPFS, decrypts it and sends it to the Buyer. (Figure 4.4)

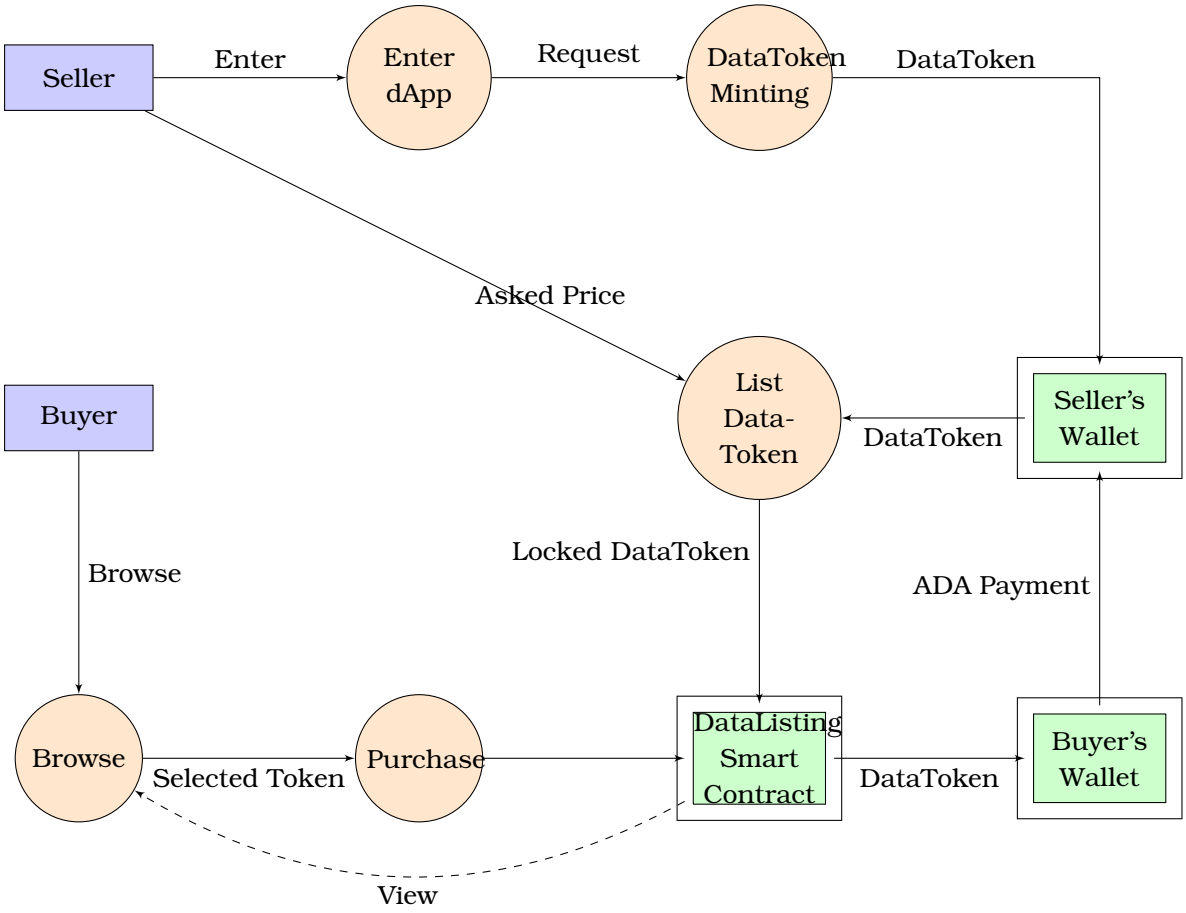

Figure 4.2. *Ask Flow process illustrating the listing and purchase of DataTokens via the DataListing Smart Contract.*

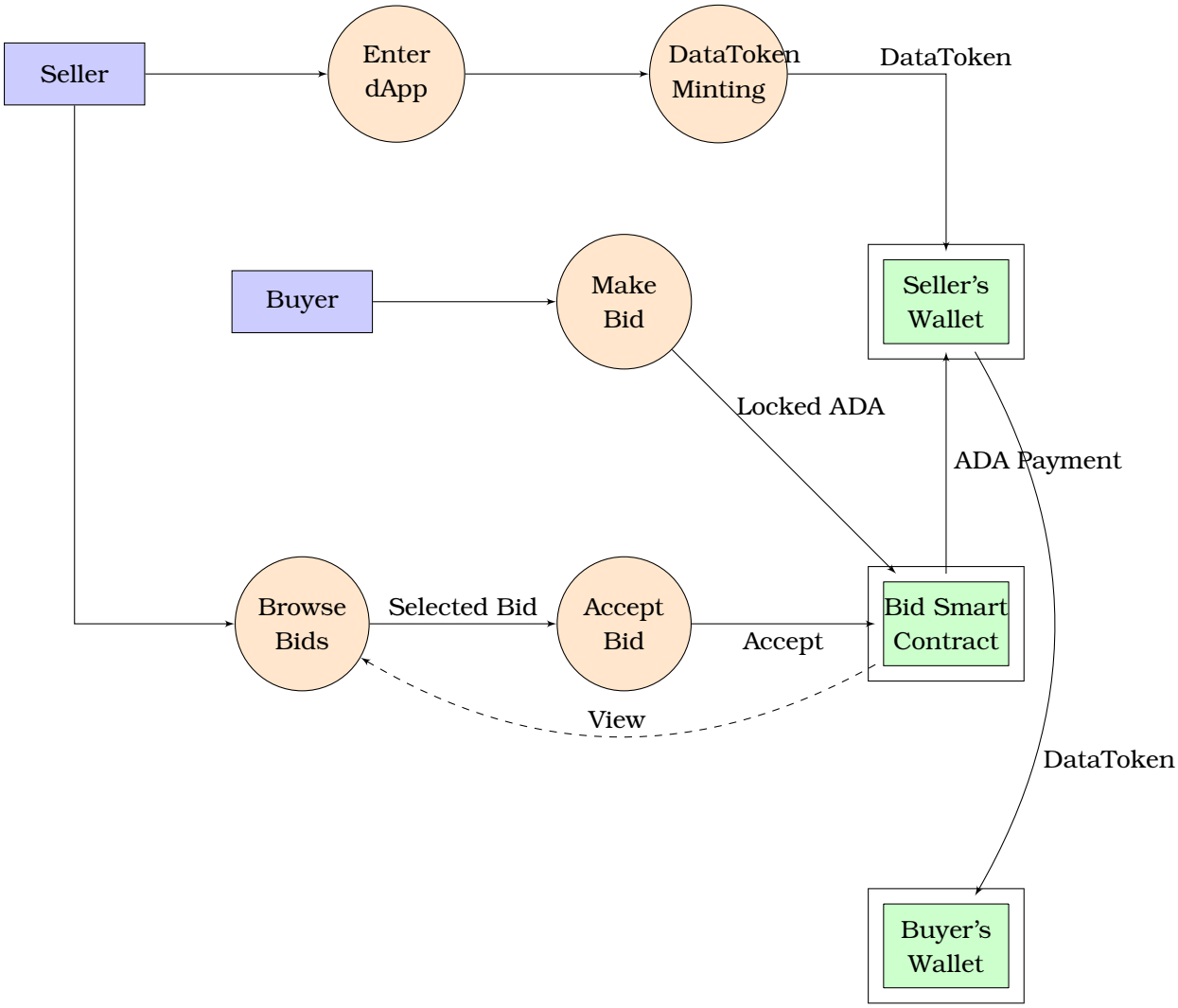

Figure 4.3. *Bid Flow diagram showing the buyer's bidding process, seller's acceptance, and the role of the Bid Smart Contract in exchanging DataTokens and ADA.*

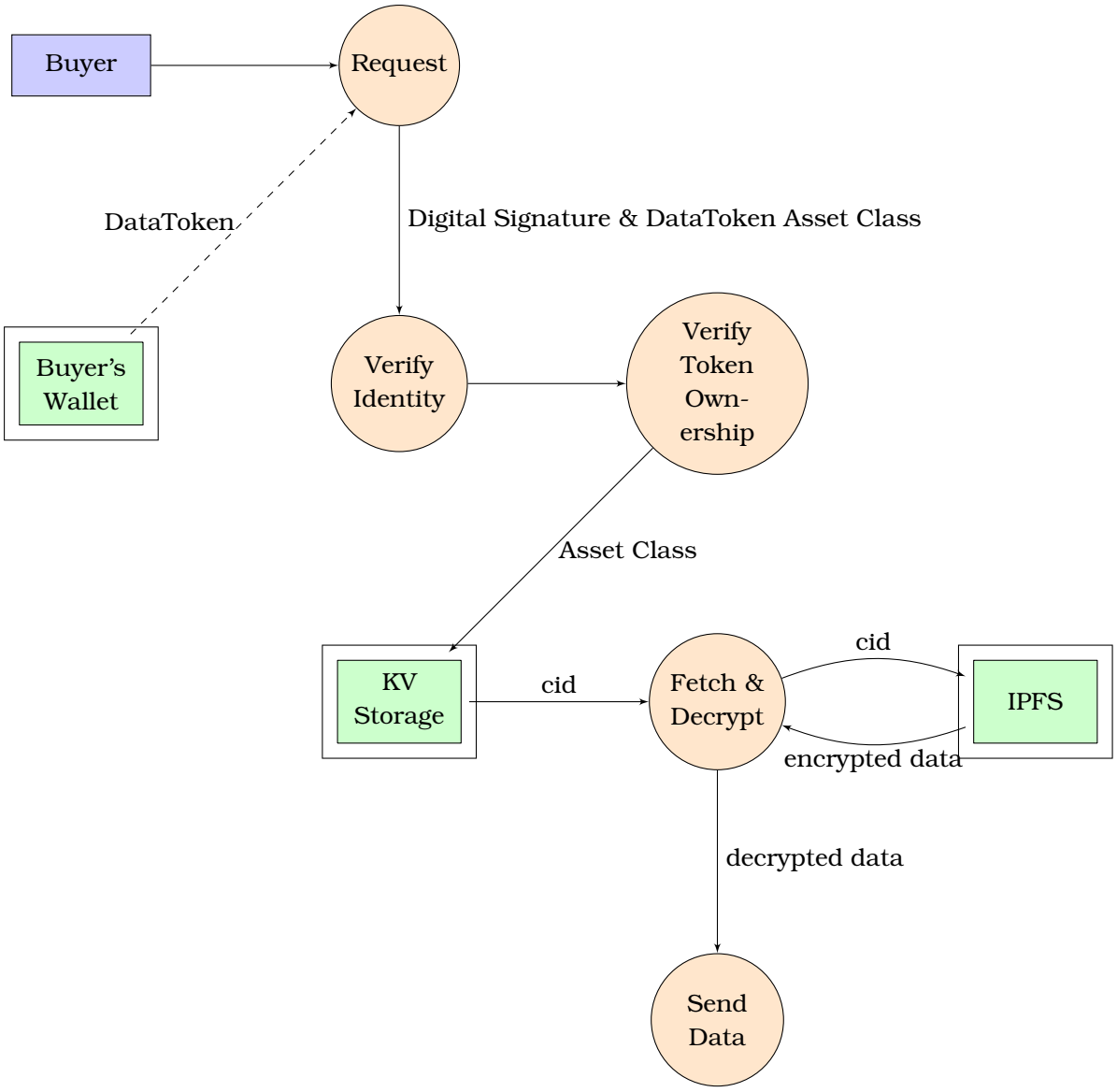

Figure 4.4. *Post-Purchase process depicting the buyer's verification and data retrieval steps following the acquisition of a DataToken.*

Chapter 5

Implementation

5.1 Front-end Development

At the heart of the front-end architecture lies the lucid-cardano library, a pivotal component that facilitates wallet interactions. With user consent, this library can access wallet information and execute transactions on behalf of the user. While the Nami wallet serves as the primary wallet for this implementation, the system is designed to be compatible with any wallet that adheres to the CIP-0030 standard, such as Yoroi, Lace, or Eternl.

In addition to wallet interactions, Blockfrost functions as the blockchain service provider. It serves as a robust alternative to running a dedicated blockchain node, offering functionalities like transaction lookup, UTXO tracking, and wallet content retrieval. Blockfrost also aids in transaction signing, thereby eliminating the need for constant synchronization with the blockchain network.

The front-end is divided into two primary interfaces: the Seller Interface and the Buyer Interface. Each interface is aimed to facilitate the two main transactional flows—'Ask' and 'Bid'—from the perspectives of the Seller and the Buyer, respectively. However, before delving into these interfaces, it's crucial to understand the role of the browser extension, exclusively utilized by sellers.

5.1.1 Browser extension

Regarding the browser extension, it has been developed using Create React App and Typescript. Given that browser extensions necessitate a build output comprising plain HTML, CSS, and JavaScript files, Webpack is employed as the bundler to meet this requirement. The extension is configured through a Manifest V3 JSON file, which outlines various entry points. The most critical entry point for this specific use-case is the popup component, with other components like contentScript, service-worker, and options page not being actively used for the extension's required functionality.

The popup component features a form with two input fields. The first field is designated for the user's wallet address, which the seller must input to later retrieve and sell their data. The second field specifies the time range, in days, for which the seller wishes to collect browsing history data; the default setting for this is seven days. Upon clicking the 'Collect' button, a method within the popup component invokes the chrome.history.search

API to gather all direct links entered by the user within the specified timeframe. After this data collection, the user can click 'Submit,' triggering the data's encryption and subsequent storage on the IPFS network.

5.1.2 dApp UI

Token Generation

Upon entering the dApp, the Seller has the option to fetch any data associated with their wallet that is stored on the IPFS network. If no data is found, a prompt advises the Seller to first use the browser extension to capture data. Otherwise, the Seller can proceed to mint their DataToken. This process also generates off-chain metadata on the server, which includes the CID of the data on the IPFS network. Additional metadata could be added in the future to provide more context to the Buyer, such as the type of data and the number of days it covers.

To mint the DataToken, the UI scans the user's wallet for an available UTXO and uses it to parameterize the Minting Policy. The resulting final minting policy is then used in a new transaction to create a unique DataToken. Upon getting the finalPolicy, we can also obtain its unique Policy ID using 'lucid.utils.mintingPolicyToId(finalMintingPolicy);'.This is essentially the hash of the final parameterized minting policy. By concatenating the Policy ID with the token's name('DataToken'), we finally obtain the token's Asset Class. This unique Asset Class acts as an identifier for the generated token.

It's worth noting that Plutus smart contracts are serialized into bytes (in hexadecimal form) to be used in off-chain code and attached to transactions.

Ask Flow

Seller After minting their DataToken, the Seller can list it for sale by creating a transaction at the DataListing contract address. The serialized DataListing smart contract is stored in React's context and its address is retrieved using lucid.utils. The Seller then creates a transaction with a datum consisting of their public key hash and asking price, locking their DataToken as the value.

Buyer The Buyer can view a table of all tokens locked under the DataListing address using lucid's utxosAt method. Since anyone can lock values with arbitrary datums at a smart contract's address, some UTxOs might be invalid. After filtering out invalid UTXOs, only the ones which adhere to our actual smart contract's Datum remain. The Buyer can select a token from one of them and proceed with the purchase.

The UTxO contains the seller's PubKeyHash, which is used for validation reasons, but we can easily obtain the Seller's address by querying our Blockfrost provider, and by using transaction hash that created the UTxO, find the address that created the UTxO and matches this PubKeyHash. The transaction pays the required amount to the Seller's address while also collecting the locked token.

When attempting to spend a UTXO that is locked by a validator, it's essential to attach the serialized validator to the transaction. This is done using the 'attachSpendingValidator' method. This off-chain check ensures that the validator will successfully unlock the UTXO before the transaction is submitted to the blockchain network.

Once the transaction meets all the conditions, including paying the required amount to the Seller, it will be successfully processed. This results in the Buyer gaining possession of the DataToken and the Seller receiving the specified amount in ADA.

After the successful transaction, the Buyer can then request the data associated with the DataToken from the server. To do this, the Buyer generates a digital signature using the lucid.wallet.signMessage method. The server verifies this signature, decrypts the corresponding data from IPFS, and returns it to the client for download.

This process ensures a secure and transparent transaction, allowing both the Buyer and the Seller to achieve their objectives while maintaining the integrity of the associated data.

Bid Flow

The Bid pattern addresses a common concern among users who are hesitant to temporarily relinquish control of their tokens, as is required in the Ask pattern. In the Bid pattern, the Seller retains possession of the token until the moment they are fully compensated.

Buyer Assuming a Seller has already minted a DataToken, the Bid pattern is initiated by the Buyer expressing interest in a specific token. Within the Buyer interface, all tokens minted by Sellers can be viewed. It's important to note that these tokens are still in the Sellers' wallets and are not locked under the DataListing contract.

The Buyer selects the token they are interested in and places a bid. To do this, the Bid smart contract is parameterized using the token's asset class, generating a final serialized form and, consequently, a unique smart contract address. The Buyer then locks a UTXO under this address, setting the bid amount as the value. This amount represents what the Seller will receive upon claiming the bid. The Buyer's public key hash is set as the datum to confirm later that the Buyer receives the token they bid for. The transaction is then signed and submitted to the network.

In the same interface, the Buyer can view all active bids they've made that have not yet been claimed by Sellers. This gives them the option to redeem their ADA if they choose to withdraw their bid. To do this, a transaction is submitted to collect the UTXO, using "Redeem" as the redeemer value. The spending validator of the Bid smart contract must be attached to the transaction, and it must be signed to match the public key hash in the datum, thereby releasing the locked ADA amount.

Seller If one or more Buyers have placed bids on a specific token, the Seller can choose which bid to claim. To do this, the Seller parameterizes the final validator with their token's asset class, generating a unique smart contract address where they can view all

bids made on their token. The Seller then selects the UTXO (bid) they wish to claim. The validator is attached to the transaction, and "Sell" is used as the redeemer value. It's crucial that this transaction also transfers the DataToken to the Buyer's wallet; otherwise, the transaction will fail. If all conditions are met, the Seller receives the bid amount in ADA, and the Buyer gains possession of the DataToken.

It's worth noting that if a Seller attempts to claim multiple bids, the transaction will fail. The smart contract ensures that each Buyer receives the specified token, and since the Seller can only possess one unit of that specific token, multiple claims are not possible.

After the transaction, the Buyer can scan their wallet for new tokens using the Buyer interface. Once the new token is detected, the UI provides the option to download the associated data. Similar to the Ask pattern, the Buyer signs the server request to validate their ownership of the token. Upon validation, the data is decrypted and sent back to the Buyer for download.

This Bid pattern offers an alternative transaction flow that addresses the concerns of users who prefer to maintain control of their tokens until the moment of sale, ensuring a secure and transparent process for both parties.

5.2 Smart Contract Development

5.2.1 DataToken Minting Policy

The DataToken Minting Policy is the first of three smart contracts utilized in this decentralized application. Unlike standard smart contracts, a minting policy is specialized to govern the creation of new tokens. By default, a minting policy receives the redeemer and the transaction's script context as arguments. However, this particular minting policy is further parameterized with a user's unique Unspent Transaction Output (UTxO), represented by a TxOutRef argument.

Each blockchain transaction has a unique transaction ID, which is derived from its hash. Additionally, each transaction can have multiple outputs, known as UTxOs, each with a distinct output index. Therefore, a UTxO can be uniquely identified within the blockchain using the format `txHash#outputIndex`.

Here's the core code for this minting policy:

```
1  mkDataTokenPolicy :: TxOutRef -> TokenName -> () -> ScriptContext -> Bool
2  mkDataTokenPolicy utxo tn () ctx = traceIfFalse "UTxO not consumed" consumesUtxo &&
3                                     traceIfFalse "Wrong amount minted" mintsExactlyOneToken
4      where
5          info :: TxInfo
6          info = scriptContextTxInfo ctx
7
8          transactionInputs :: [TxInInfo]
9          transactionInputs = txInfoInputs info
10
11         consumesUtxo :: Bool
12         consumesUtxo = any (\i -> txInInfoOutRef i == utxo) transactionInputs
```

```
13
14          valueMinted :: Value
15          valueMinted = txInfoMint info
16
17          mintsExactlyOneToken :: Bool
18          mintsExactlyOneToken = case flattenValue valueMinted of
19              -- we ignore currencySymbol
20              [(_, tn', amt)] -> tn' == tn && amt == 1
21              _               -> False
```

The DataToken Minting Policy is designed to enforce two critical rules:

1. UTxO Consumption: The minting policy verifies that the transaction consumes the specific UTxO with which the policy was parameterized. This ensures that only the owner of that UTxO can mint a new DataToken. This is crucial for maintaining the integrity and uniqueness of each minted token.

2. Single Token Minting: The policy also checks that only one DataToken is minted in the transaction. This is a domain-specific rule for the dApp, ensuring that each DataToken represents a unique set of data and maintains its individual value.

5.2.2 DataListing Smart Contract for the Ask Flow

The DataListing smart contract serves as the validator for the Ask flow, where a seller locks their DataToken and sets an asking price for it. This smart contract is a Validator, which means it takes three arguments: the redeemer, the script context, and the Datum.

```
1  data DataListDatum = DataListDatum
2      {
3       dataSeller  :: PubKeyHash
4      , price      :: Integer
5      } deriving Prelude.Show
6
7  data DataListingRedeemer = Redeem | Purchase
8
9
10 mkValidator :: DataListDatum -> DataListingRedeemer -> ScriptContext -> Bool
11 mkValidator dat r ctx = case r of
12     Purchase -> traceIfFalse "Amount required not paid to owner" buyerHasPaidSeller &&
13         traceIfFalse "You must consume only one utxo" consumesOnlyOneUtxo
14     Redeem   -> traceIfFalse "data seller's signature missing" checkSignedBySeller
15
16   where
17
18     info :: TxInfo
19     info = scriptContextTxInfo ctx
20
21     -- In lovelaces
22     dataPrice :: Integer
```

```
23      dataPrice = price dat
24
25      valuePaidToSeller :: Value
26      valuePaidToSeller = valuePaidTo info (dataSeller dat)
27
28      pricePaidToSeller :: Integer
29      pricePaidToSeller = valueOf valuePaidToSeller adaSymbol adaToken
30
31      buyerHasPaidSeller :: Bool
32      buyerHasPaidSeller = pricePaidToSeller >= dataPrice
33
34      consumesOnlyOneUtxo = length consumedInputsOfThisScript == 1
35          where
36          scriptAddress :: ValidatorHash
37          scriptAddress = ownHash ctx
38
39          -- The credentials that are required to unlock each input, can be either PubKeyHash,
40          -- which means they belong to a user, and they unlock them by signing with their pk,
41          -- or ScriptCredentials, that require the script to be included, and validated
42          inputScriptCredentials :: [Credential]
43          inputScriptCredentials = map (addressCredential . txOutAddress . txInInfoResolved) $ txInfoInputs info
44
45          consumedInputsOfThisScript = filter protectedByThisScript inputScriptCredentials
46              where
47              protectedByThisScript :: Credential -> Bool
48              protectedByThisScript c = case c of
49                  PubKeyCredential _ -> False
50                  ScriptCredential vh -> vh == scriptAddress
51
52      checkSignedBySeller :: Bool
53      checkSignedBySeller = txSignedBy info $ dataSeller dat
```

Redeemer Capabilities The DataListing smart contract is designed to accommodate two different actors:

1. The Buyer: Who wishes to purchase the DataToken.
2. The Seller: Who may want to cancel the DataListing and retrieve their DataToken.

When the "Purchase" redeemer is invoked, the validator performs two critical checks:

1. Payment Verification: It verifies that the buyer has paid the seller the agreed-upon amount, as specified in the Datum. This ensures that the transaction is fair and aligns with the seller's asking price.
2. Single Token Unlocking: It checks that the spending transaction attempts to unlock only one DataToken. This is a crucial security measure to prevent potential vulnerabilities.

- Security Implication: Without this check, a vulnerability could arise. Consider a scenario where two DataTokens from the same seller are locked under this smart contract, both requiring the same price (e.g., 40 ADA) in the Datum. A malicious buyer could craft a transaction that pays the seller 40 ADA but attempts to unlock both tokens. This check effectively mitigates such risks.

5.2.3 Bid Smart Contract for the Bid Flow

The Bid smart contract serves as the backbone for the Bid flow, where a buyer places a bid on a DataToken. This is a parameterized validator, meaning it takes four arguments: the token it has been parameterized with, the Datum, the Redeemer, and the Script Context.

```
data BidParams = BidParams
    {
     dataTokenNFT :: AssetClass
    }

data BidDatum = BidDatum
    {
     dataBuyer   :: PubKeyHash
    } deriving Prelude.Show


data BidRedeemer = Redeem | Sell

mkValidator :: BidParams -> BidDatum -> BidRedeemer -> ScriptContext -> Bool
mkValidator p dat r ctx = case r of
          Sell   -> traceIfFalse "token not given to buyer" buyerGetsToken
          Redeem -> traceIfFalse "buyer's signature missing" checkSignedByBuyer

  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    checkSignedByBuyer :: Bool
    checkSignedByBuyer = txSignedBy info $ dataBuyer dat

    valuePaidToBuyer :: Value
    valuePaidToBuyer = valuePaidTo info $ dataBuyer dat

    buyerGetsToken :: Bool
    buyerGetsToken = assetClassValueOf valuePaidToBuyer (dataTokenNFT p) == 1
```

Checks and Constraints The Bid smart contract performs checks that are conceptually similar to those in the DataListing contract, but with some minor differences:

1. No Double Spending: Since each DataToken can only be minted once, there's no

risk of double spending in this context. This simplifies the validation logic compared to the DataListing contract.

2. Single Bid per Token: The contract is designed such that a buyer should not place multiple bids on the same DataToken. If a buyer wishes to change their bid, they should first redeem their existing bid before placing a new one.

    - Security Note: If a scenario arises where placing multiple bids is desired, an additional check could be implemented to ensure that only one UTxO is consumed. This would help prevent any potential double-spending vulnerabilities.

This completes the overview of the three smart contracts powering the dApp: DataToken Minting Policy, DataListing, and Bid. Each has its own set of checks and constraints to ensure secure and fair transactions within the marketplace.

5.3 Back-end Development

The backend plays an important role in encrypting and storing the user's data, and also storing off-chain metadata for the tokens, which can then be later used by the buyers to retrieve their purchased assets. It also has some validation mechanisms to validate the requests and verify the token owners are making the requests. This ensures that only the token holders can decrypt and download the browsing data. It also interacts with the IPFS network, where the data is saved in an encrypted form.

5.3.1 API Routes

Route for Token & Data Association

`saveHistory` Route

Endpoint: `/api/saveHistory`

Purpose: This is the first route typically invoked in the user flow. It is triggered by the browser extension and receives the user's wallet address and browsing data in its request body.

Functionality: The server encrypts the received browsing data and stores it on the IPFS network. It then associates the returned CID (Content Identifier) with the user's wallet address. A key-value database is used for both authentication and token metadata storage.

Validation: The server validates the request body to ensure it contains a valid wallet address and browsing data.

The user then, when entering the dApp, which uses Lucid to connect to the user's wallet, makes a request to the server to retrieve any data associated with this wallet.

`retrieveHistory` Route

Endpoint: `/api/retrieveHistory`

Purpose: This route is invoked when the user accesses the dApp, which connects to the user's wallet using the Lucid library.

Functionality: The server retrieves any data associated with the user's wallet.

Validation: A digital signature is used to validate that the request is indeed being made by the owner of the wallet.

`associateDataWithToken` Route

Endpoint: `/api/associateDataWithToken`

Purpose: This route is responsible for associating a newly generated token's asset class with the previously stored CID, effectively creating off-chain metadata for the token.

Functionality:

- Associates the CID with the new token's asset class.
- Deletes the previous association of the CID with the user's wallet.
- Creates a TokenListing for tokens that are available but not yet locked under a DataListing.

Validation:

- Verifies that the token's asset class and CID are valid.
- Ensures that the previous wallet owns the new token.

Routes for Updating Token Listings and Active Bids The backend architecture is designed not only to manage the tokens and their associated data but also the listings and active bids for these tokens. Below are the key routes and functionalities for managing token listings and active bids:

Token Listings

Token listings are created for tokens that are available for purchase but have not been locked under a DataListing contract. Their functionality is to allow buyers to browse through available tokens and place bids on the ones they are interested in.

`fetchAll` Route:

Endpoint: `/api/tokenListing/fetchAll`

Purpose: To retrieve all available token listings so that bidders can browse through them.

Functionality: Returns a list of all tokens that are available but not yet locked under a DataListing contract.

`deleteTokenListing` Route:

Endpoint: `/api/tokenListing/delete`

Purpose: To remove a token listing once a bid has been approved, and the token is no longer available for sale.

Functionality: Deletes the specific token listing based on its unique identifier.

Active Bids

Active bids are created when buyers place bids on available token listings. The backend manages these bids until they are either redeemed or fulfilled.

`createActiveBid` Route:

Endpoint: `/api/activeBid/create`

Purpose: To create a new active bid when a buyer places one.

Functionality: Adds a new entry to the collection of active bids, storing essential information like transaction hash, amount, token asset class, and date.

`deleteActiveBid` Route:

Endpoint: `/api/activeBid/delete`

Purpose: To remove an active bid, typically when it has been redeemed or fulfilled.

Functionality: Deletes the specific active bid based on its unique identifier.

`fetchActiveBidsByWallet` Route:

Endpoint: `/api/activeBid/fetchByWallet`

Purpose: To retrieve all active bids associated with a specific wallet.

Functionality: Returns a list of all active bids tied to a particular wallet.

5.3.2 Data Management with IPFS

The InterPlanetary File System (IPFS) serves as the data storage layer for the dApp, specifically for storing encrypted user browsing history. IPFS offers a decentralized approach to data storage, making it an ideal fit for this application.

Using Blockfrost as an IPFS Gateway

While it's possible to deploy and manage a custom IPFS node, leveraging a service provider like Blockfrost simplifies the process. Blockfrost, already in use as a blockchain provider, can also act as an IPFS gateway, handling the complexities of data storage and retrieval.
Below is a simplified example of how this can be achieved in Nodejs using Typescript.

```
import { BlockFrostIPFS } from '@blockfrost/blockfrost-js';

const IPFS = new BlockFrostIPFS({
      projectId: ipfsProjectId,
    });
const added = await IPFS.add(tmpFilePath);

// Pin the data to ensure it remains accessible
const pinned = await IPFS.pin(added.ipfs_hash);
const cid = pinned.ipfs_hash;
```

Listing 5.1: Store data in IPFS using Blockfrost

Pinning Mechanism Pinning is a crucial feature in IPFS that ensures the persistent storage of data objects. When data is first uploaded to IPFS via Blockfrost, it should be pinned to ensure it remains accessible and isn't subject to garbage collection. This needs to be done frequently to avoid being garbage-collected. In a more advanced implementation, a cron job could be set up to regularly re-pin data, thereby ensuring its long-term availability.

By integrating IPFS and using Blockfrost as a gateway, the backend architecture achieves a robust, decentralized data storage solution. This setup not only aligns with the decentralized nature of the dApp but also offers a scalable and secure way to manage user data.

By employing these routes and their associated validation mechanisms, the backend ensures secure and efficient management of tokens and their associated data.

5.4 Authorization Mechanisms

5.4.1 Digital Signature for Identity Verification

In the world of blockchain-based dApps, digital signatures are crucial for verifying a user's identity. This process employs asymmetric cryptography, where each user has a pair of keys: a public key that is publicly shared, and a private key that remains confidential. When a user wishes to send a message or conduct a transaction, they sign it with their private key. This signature serves as a permanent immutable stamp of their identity.

Signature Generation The user employs their private key to create a digital signature on the data or message. This is usually done by taking a hash of the message and encrypting

it using the private key. The digital signature is then appended to the message. In this dApp, this is achieved client-side using the `lucid-cardano` library as follows:

```
const signature = lucid.wallet.signMessage(wallet, hexPayload);
```

Listing 5.2: Signature Generation

Signature Verification To verify the sender's identity, the recipient uses the sender's public key to decrypt the attached digital signature. They then hash the received message and compare it to the decrypted hash. If both hashes match, it confirms that the message has not been tampered with and indeed originates from the owner of the public key. This verification is done server-side, also using `lucid-cardano` library:

```
const signatureIsValid = lucid.verifyMessage(wallet, hexPayload, signature);
```

Listing 5.3: Signature Verification

This mechanism ensures both the integrity and the origin of the message, serving as a foundational element for secure and trustless communications within the dApp.

Time-Stamping in Digital Signatures Including a timestamp in a digital signature is a common practice, this technique is known as "time-stamping" and serves multiple purposes:

- Expiration: By including a timestamp, we can set an expiration time for the digital signature, making it invalid after a certain period. This can be useful for temporary authorizations or time-sensitive transactions.
- Non-repudiation: The timestamp provides additional evidence for the exact time when the document was signed, which can be crucial in legal and financial applications.
- Security: If a private key is compromised, the timestamp can help limit the time frame during which the compromised key was used to sign documents, aiding in damage control.

5.4.2 Seller Authorization

Ask and Bid Flow The seller initiates the data collection process via a browser extension, where they are required to input their Cardano wallet address. The collected data is then sent to the backend, encrypted, and stored on IPFS.

Additionally, an association between the wallet address or token asset class to the IPFS CID (Content Identifier) is stored in the server's key-value database. This approach is similar to attaching on-chain metadata on the token, by including metadata on the transaction that minted the token.

Upon entering the Next.js dApp, the application connects to the seller's wallet using the `lucid-cardano` library. This method is compatible with any CIP-0030 compliant wallet, such as Nami, Eternal, or Yoroi. The dApp then fetches any data associated with the

seller's wallet from the server, and proves the user's identity by also sending a digital signature. The server verifies this signature and returns the data to the Seller for inspection and token minting.

5.4.3 Buyer Authorization

The Buyer initiates a request to the server with the aim of decrypting and downloading the data linked to a specific token they have purchased. To do this, the buyer sends both the token's asset class and their own wallet address. The server then proceeds on a two-step verification process:

1. Identity Verification: Similar to the mechanism used for the Seller, the server first confirms that the request is genuinely coming from the owner of the provided wallet address. This is achieved through digital signature verification.

2. Token Ownership Verification: Utilizing the Blockfrost provider, the server checks that the wallet in question indeed holds at least one UTxO associated with the specified token asset class.

Security Implications of Skipping Verification If the server was to bypass these verification steps, it would open the door for potential security risks. Specifically, any user who knows a token's asset class could falsely claim ownership and request access to the data. This would fundamentally undermine the trust and integrity of the entire system.

Authorization Mechanisms of TokenListings and ActiveBids

While wallet validation could also be used for these routes, an alternative and potentially more efficient way to authorize these requests could be to send the corresponding transaction hash related to each action. The server could then validate this using a blockchain provider to ensure the input data is correct. For instance, an active bid creation request should send the txHash of the transaction that locked the UTxO under the Bid's address.

5.5 Testing in Plutus: Ensuring Smart Contract Integrity

In the realm of decentralized applications (dApps), the role of testing is not merely a best practice but an imperative. Given the immutable nature of blockchain transactions, errors and vulnerabilities can have irreversible consequences. This section mentiones common testing methodologies employed in Plutus, to ensure the integrity and robustness of smart contracts.

5.5.1 Unit Testing: The First Line of Defense

Unit testing is a software testing technique where individual units or components of a software are tested in isolation to ensure that they function as intended, hence they are very handy for testing smart contracts.

The Plutus simple model library is specifically designed for unit testing, and it employs state monads to simulate the blockchain environment. The state monad serves as a wrapper around the blockchain state, allowing for a sequence of transactions to be tested for validity. This is particularly useful for testing various spending scenarios, including normal spending and double-spending attempts.

Here is a snippet from DataListing's unit tests, that is designed to catch the double-spending vulnerability discussed previously when analyzing the smart contracts:

```
1  -- This method tries to double spend, and runs successfully if the bad actor manages to double spend
2  -- If they fail to exploit, this method logs an error
3  doubleSpending :: Run ()
4  doubleSpending = do
5    (u1,u2) <- setupUsers
6
7    -- Lock 2 tokens utxos, asking for 400 each
8    let token = fakeValue scToken 1
9    sp1 <- spend u1 token
10   submitTx u1 $ lockingTx u1 400 sp1 token
11
12   sp2 <- spend u1 token
13   submitTx u1 $ lockingTx u1 400 sp2 token
14   --
15   -- Get the locked utxos
16   scriptsUtxos <- utxoAt dataListingScript
17   let ((ref1, out1), (ref2, out2)) = scriptsUtxos
18
19   let buyerPaying = adaValue 400
20   u2_sp <- spend u2 buyerPaying
21   submitTx u2 $ doubleConsumingTx u2 u1 u2_sp buyerPaying (ref1,ref2) (txOutValue out1) (OnChain.DataListDatum u1 400)
22
23   (v1,v2) <- mapM valueAt (u1,u2)
24   -- If user 1 has only 400, this succeeds(logError does not run), and since it's a bad test, it will fail
25   unless (v1 == adaValue 400 &&
26     v2 == adaValue 600 <> fakeValue scToken 2)
27     $ logError $ "Error occured. Received values: " ++ show (fmap flattenValue (v1,v2))
```

In this test, the function doubleSpending is designed to simulate a scenario where a cunning Buyer attempts to double-spend by purchasing 2 tokens, while paying only for 1. In this scenario the Seller(u1) has some tokens, and the Buyer(u2) has 1000 ADA. The seller proceeds to list his 2 tokens for 400 ADA each. The abuser would try to consume both for 400 ADA instead of 800, so if they succeed, they expect that User1 will have 400 ADA, while they will have 600 ADA and 2 tokens. So we write the method from the abuser's perspective, and later we define this test case as "bad", expecting it to fail and log an error, in order for our test to pass.

These kinds of tests ensure that the smart contract logic correctly identifies and prevents this malicious activity, thereby safeguarding the integrity of the transactions.

Property Testing: Beyond Unit Tests While unit tests are excellent for catching specific errors, they are not sufficient for proving the overall reliability of a smart contract. Plutus also supports property testing through the Haskell library QuickCheck, which automatically generates test cases to validate the properties of the smart contract. This adds an additional layer of assurance that the contract behaves as expected under a wide range of conditions.

Testing is an indispensable component in the development lifecycle of Plutus smart contracts. Through a combination of unit tests and property tests, developers can ensure that the smart contracts are both secure and functional, thereby fostering trust and reliability in dApps built on the Cardano blockchain.

Part III

Analysis

Chapter 6

Results and Discussion

6.1 The Transformative Potential of Smart Contracts

One of the most striking revelations during the course of this project was the the remarkable flexibility and transformative potential of smart contracts. In traditional systems, the process of data exchange and financial transactions often relies on centralized intermediaries, which can be both costly and less secure. Smart contracts, however, offer a paradigm shift in how we can automate these processes in a decentralized, trustless environment.

In the context of this project, smart contracts took control of the payout process, effectively eliminating the need for a trusted third party to oversee transactions. This not only reduces the risk of fraud but also streamlines the entire process, making it more efficient and transparent.

The trustless nature of smart contracts is particularly revolutionary. By encoding the rules of the transaction within the contract itself, we can create a self-executing and self-enforcing agreement that neither party can tamper with once deployed. This has profound implications for a wide range of applications beyond just data marketplaces, from supply chain management to decentralized finance (DeFi).

This project serves as a testament to some of the capabilities of smart contracts, and it opens the door to rethinking how we approach not just data transactions, but a multitude of processes that can benefit from decentralization and automation without sacrificing security.

6.1.1 Technical Achievements and Challenges

Mastery of Haskell and Plutus One of the most significant milestones in this project was gaining proficiency in Haskell and Plutus. While I have a strong background in functional programming languages like Standard ML (SML), Haskell presented unique challenges, particularly in understanding advanced concepts like Monads. The Plutus Pioneer Program from IOHK was instrumental in bridging this gap, providing essential insights into developing on the Cardano Network. Resources like the book "Learn You a Haskell for Great Good!", which i read at least twice, significantly aided in reducing Haskell's steep learning curve.

Browser Extension Development My initial plan to encapsulate the entire marketplace within a browser extension had to be revised due to its limitations. These limitations became evident during development, as I gained a better understanding of the architecture and security measures dictating browser extensions. The extension's role was narrowed down to data collection, making the system more robust and focused.

Smart Contract Security The implementation and testing of smart contracts were crucial in ensuring the integrity of the marketplace. Catching vulnerabilities and confirming their mitigation in live scenarios, such as the double-spending case, were significant achievements.

6.1.2 Future Directions

Enhanced Data Collection The current data collection is a proof of concept and can be significantly enriched. By integrating analytics libraries and obtaining user consent, the system could capture more nuanced data like user navigation patterns and engagement heat-maps.

Smart Contract Improvements Several improvements can be made to the smart contracts, such as deploying reusable scripts to reduce costs and adding time-based conditions for more complex interactions. Furthermore, the current implementation is limited to a single data asset per transaction, which could be optimized to allow for multiple data assets per transaction.

Real-Time Updates in dApp The dApp could be made more dynamic by incorporating real-time updates through web sockets or server-sent events.

Client-Side Encryption Exploring client-side encryption of the collected data could further decentralize the dApp. However, this poses challenges as storing secrets on the blockchain is not advisable. This remains an area for future research.

Fee Optimization The current implementation could learn from JPG Store's bid flow, which allows for more asset transfers per transaction, optimizing the fee per data asset paid.

This implementation has the potential to revolutionize how user data is captured and valued on the web. Currently, users have little incentive to agree to cookie consents; this system could provide them with tangible benefits, thereby changing the dynamics of data consent and collection.

Chapter 7

Ethical Considerations

Data privacy is a critical concern in the development and deployment of decentralized applications (dApps), especially those that handle sensitive user information. This section explores the ethical considerations surrounding data privacy in the context of dApps, with a focus on this specific case of a dApp that stores and sells encrypted seller data on the InterPlanetary File System (IPFS).

Informed Consent Before storing any seller data, it's crucial to obtain informed consent from the users. The browser extension would be the ideal place to obtain informed consent, especially since it's the point of interaction where the data is initially captured. When the user installs the extension or when they first use the feature that captures browser history, a clear and easily understandable consent form should pop up.

According to the General Data Protection Regulation (GDPR), consent is defined as "any freely given, specific, informed and unambiguous indication of the data subject's wishes by which he or she, by a statement or by a clear affirmative action, signifies agreement to the processing of personal data relating to him or her" [19]. This involves clearly explaining what data will be stored, how it will be used, and who will have access to it. Informed consent is not just an ethical requirement but also a legal one under regulations like the GDPR.

Encryption Encrypting the data before storing it on IPFS adds an extra layer of security, making it difficult for unauthorized users to access the information. However, encryption is not a silver bullet and it should be part of a broader data protection strategy. Vulnerabilities can arise from weak algorithms, poor key management, human error, and brute force attacks [20].

Anonymization Whenever possible, storing the data anonymized on the IPFS network is a good practice to protect the identity of the individuals involved.

Data Retention Policy A clear data retention policy should be in place, specifying how long the data will be stored and what will happen to it after that period. This is particularly important for complying with the "right to be forgotten" under GDPR's Article 17.

IPFS and Data Privacy Storing data on IPFS comes with its own set of challenges and opportunities. On the one hand, IPFS's decentralized nature eliminates the risks associated with centralized data storage systems. On the other hand, once data is on IPFS, it's there permanently unless steps are taken to enable its removal. It's worth noting that data not regularly pinned can be subject to garbage collection as per IPFS protocol. This means that if you want to ensure the longevity of your data, you must actively manage its pinning. Instead, explicit unpinning can be a method to facilitate data removal, although this doesn't guarantee that other nodes haven't already replicated the data.

Data privacy is a varied issue that requires a comprehensive approach, including informed consent, encryption, anonymization, and a clear data retention policy. Special considerations are needed when using technologies like IPFS, which come with their own sets of benefits and drawbacks.

Part IV

Epilogue

As I conclude this thesis, it's worth taking a moment to reflect on the journey that led to its completion. The process of developing a decentralized application for data marketplace has been both challenging and enlightening, offering numerous lessons that extend beyond the technical realm.

The work presented in this thesis is not just a technical exercise but a step towards rethinking how we approach data privacy and ownership in the digital age. By leveraging blockchain technology and smart contracts, we can envision a future where control over personal data is returned to the individual, disrupting traditional models of data monetization and consent.

The challenges encountered during this project, from mastering Haskell and Plutus to navigating the complexities of decentralized systems, have been invaluable learning experiences. They have not only honed my technical skills but also deepened my understanding of the ethical and societal dimensions of technology. As for what comes next, whether it's further academic research or diving into industry applications, I am excited to continue exploring the transformative potential of blockchain technology.

Advice for Future Researchers

For anyone embarking on a similar journey, my advice would be to embrace the challenges as opportunities for growth. The rapidly evolving landscape of blockchain technology offers a fertile ground for innovation but also requires a willingness to adapt and learn continuously.

Closing Thoughts

In a world increasingly driven by data, the need for secure, transparent, and just systems for data exchange is more pressing than ever. This thesis serves as a testament to the possibilities that emerge when technology is aligned with these principles. While there is still much work to be done, both in refining the current system and exploring new applications, the path forward is promising.

Source Code

The code can be accessed on GitHub at https://github.com/varagos/data-sail.

Bibliography

[1] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008. Available online.

[2] M. N. Bhutta, A. Khwaja, A. Nadeem, H. F. Ahmad, M. Khan, M. Hanif, H. Song, M. A. Alshamari και Y. Cao. *A Survey on Blockchain Technology: Evolution, Architecture and Security*. *IEEE Access*, 2021. Available as PDF.

[3] H. Liu, R. G. Crespo και O. S. Martínez. *Enhancing Privacy and Data Security across Healthcare Applications Using Blockchain and Distributed Ledger Concepts*. 2020.

[4] Nicola Atzei, Massimo Bartoletti, Stefano Lande και Roberto Zunino. *A formal model of Bitcoin transactions*, 2017. Paper 2017/1124.

[5] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. Peyton Jones και P. Wadler. *The Extended UTXO Model*. 2020.

[6] L. Brünjes, M. Gabbay και others. *UTxO- vs account-based smart contract blockchain programming paradigms*. 2020.

[7] Wikipedia. *Halting problem*. https://en.wikipedia.org/wiki/Halting_problem. Accessed: 18-10-2023.

[8] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, J. Müller, M. Peyton Jones, P. Vinogradova και P. Wadler. *Native Custom Tokens in the Extended UTXO Model*, Year.

[9] *Plutus Documentation - Parameterized contracts*. https://plutus-pioneer-program.readthedocs.io/en/latest/pioneer/week3.html#example-2-parameterized-contract. Accessed: 18-10-2023.

[10] IOHK. *The Plutus Platform Technical Report*. https://ci.iog.io/build/1231235/download/1/plutus.pdf. Accessed: 13-10-2023.

[11] Cardano Documentation. *Collateral Mechanism*. https://docs.cardano.org/plutus/collateral-mechanism/. Accessed: 13-10-2023.

[12] M. M. Wachs, M. Hoque και M. Vukolić. *Decentralized Applications: The Blockchain-Empowered Software System*. 2018.

[13] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. Original IPFS white paper.

[14] D. Trautwein, A. Raman, G. Tyson, I. Castro, W. Scott, M. Schubotz, B. Gipp και Y. Psaras. *Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web*. *Association for Computing Machinery*, 2022.

[15] Joshua Stone. *Book.io Whitepaper*, 2023. Available online.

[16] Google. *Content Security Policy (CSP)*. https://developer.chrome.com/docs/extensions/mv3/manifest/content_security_policy/. Accessed: 13-10-2023.

[17] Google. *Extensions 101*. https://developer.chrome.com/docs/extensions/mv3/getstarted/extensions-101/. Accessed: 17-10-2023.

[18] Google. *Architecture overview*. https://developer.chrome.com/docs/extensions/mv3/architecture-overview/.

[19] European Union. *GDPR Article 4(11)*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679, 2016.

[20] M. Mohan, M. Devi και V. Prakash. *Security Analysis and Modification of Classical Encryption Scheme*. https://www.ijstr.org/final-print/apr2015/Security-Analysis-And-Modification-Of-Classical-Encryption-Scheme.pdf, 2015.

List of Abbreviations

UTxO	Unspent Transaction Output
EUTxO	Extended Unspent Transaction Output
dApp	Decentralized Application
API	Application Programming Interface
IPFS	InterPlanetary File System
P2P	Peer-to-Peer
HTTP	Hypertext Transfer Protocol
DFD	Data Flow Diagram
KV	Key-Value