

A DECENTRALIZED ETHEREUM PLATFORM FOR SMART ENERGY TRADING

Designing and implementing an eAuction application for energy trading on the Ethereum blockchain by using smart contracts.

Written by

CRISTINA TORP

EVEN EILERTSEN

SUPERVISOR

Harsha Sandaruwan Gardiyawasam Pussewalage

Co-Supervisor

Vladimir Oleshchuk

University of Agder, 2021

Faculty of Engineering and Science

Department of Information and Communication Technology

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	Vi erklærer videre at denne besvarelsen: <ul style="list-style-type: none">• Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands.• Ikke refererer til andres arbeid uten at det er oppgitt.• Ikke refererer til eget tidligere arbeid uten at det er oppgitt.• Har alle referansene oppgitt i litteraturlisten.• Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Ja

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).

Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

First and foremost, we would like to express our gratitude to Dr. Harsha S. Gardiyawasam. Without his guidance, this thesis would not have been written. The continuous meetings and feedback that we received during the last month of the semester guided our focus into what we believe resulted in a significant improvement. We would also like to thank Professor Vladimir A. Oleshchuk for his assistance during the beginning of the project. Last but not least, we would like to thank our partners for their invaluable support during this thesis.

Abstract

The modern energy grid is constantly improving its efficiency and flexibility by adopting new technology. Regional energy providers, however, have a monopolistic role in deciding market prices, and their motives have been criticized for focusing on their own profit. Microgrids have seen a large adoption partly due to their ability to supplement a local grid with alternative renewable energy sources. A decentralized auction platform would allow the users to trade energy within their local microgrids. The platform could also allow households with energy production means like solar panels to sell their excess energy, reducing the regional providers' and intermediaries such as energy brokers' role in the current grid system while improving green energy utilization.

This thesis proposes an auction application developed on the blockchain protocol Ethereum. The proposed solution would allow users to buy and sell energy using the underlying Ethereum network, making it a decentralized trading platform. The platform's security has been addressed by prioritizing security throughout the process; from the initial design phase to the creation of the solution prototype. Furthermore, quality assurance of the solution was evaluated with unit tests that addressed the implementation code, and a security analysis of the solution was conducted with respect to the users' security requirements.

Contents

Acknowledgements	i
Abstract	iii
List of Figures	ix
List of Code Fragments	xii
1 Introduction	1
1.1 Microgrids	1
1.2 Smart Energy Trading	2
1.3 Blockchain Technology	3
1.4 Motivation for Blockchain-based Energy Trading	3
1.5 Thesis Definition	4
1.5.1 Research Objectives	4
1.5.2 Research Questions	4
1.5.3 Scope	5
1.6 Related Work	5
1.6.1 Smart Grids	5
1.6.2 Blockchain-based Trading Systems	6
1.7 Contribution	7
1.8 Thesis Outline	8
2 Theoretical Background	11
2.1 Blockchain Technology	11
2.1.1 Blockchain History	11

2.1.2	Blockchain networks	12
2.1.3	Features of Blockchain	14
2.2	Ethereum	15
2.2.1	Ethereum Whitepaper and Core Ideas	15
2.2.2	Ecosystem of Ethereum	16
2.2.3	Smart Contracts	17
2.3	Cryptographic Primitives in the Platform	18
2.3.1	Hash Functions	18
2.3.2	Digital signatures	18
2.4	Blockchain Development	19
2.4.1	Solidity	19
2.4.2	Truffle	22
2.4.3	Ganache	23
3	Blockchain-based eAuction Solution	25
3.1	Case Description	25
3.2	Solution Overview	28
3.2.1	Phase 1: Create Auction	30
3.2.2	Phase 2: Hidden Round	31
3.2.3	Phase 3: Open Round	32
3.2.4	Phase 4: Close Auction	34
3.2.5	Phase 5: Delete Auction	35
4	Implementation	37
4.1	Platform Architecture	37
4.1.1	Actors and Platform Components	39
4.2	Code Implementation	40
4.2.1	Auction Controller State Variables	40
4.2.2	Auction State Variables	41
4.2.3	Phase 1: Create Auction	46
4.2.4	Phase 2: Bid in the Hidden Round	49
4.2.5	Phase 3: Bid in the Open Round	52
4.2.6	Phase 4: Close Auction	55
4.2.7	Phase 5: Delete Auction	61

5	Implementation Assessment	65
5.1	Setup	65
5.1.1	Testing Third Party Libraries	67
5.2	Auction Controller Tests	67
5.3	Auction Tests	73
5.3.1	Tests During Hidden Round	76
5.3.2	Tests During Open Round	79
5.3.3	Tests For Closing The Auction	84
5.4	Security Analysis	88
5.4.1	Attack Vectors	88
6	Conclusion	91
6.1	Future Work	93
	Appendix A Auction Controller Source Code	95
	Appendix B Auction Source Code	97
	Appendix C Auction Controller Tests Source Code	106
	Appendix D Auction Tests Source Code	113
	Bibliography	127

List of Figures

2.1	Simplified blockchain	12
2.2	Transactions in Bitcoin	13
2.3	Ganache UI	23
3.1	Overview of desired system	26
3.2	Overview of the proposed solution	29
3.3	Seller creates auction	30
3.4	Bidder(s) transfer their deposit bid with their hashed bid	31
3.5	Bidder(s) reveal their bid	33
3.6	The DApp finds the auction winner, and completes the closing logic of the auction	34
4.1	System architecture	38
4.2	UML-symbols of the actors and components included in the Smart Trading Platform implementation	39
4.3	Phase 1: No state (creation)	46
4.4	Model phase 2: Ready for hidden bids	49
4.5	Model phase 3: Ready for open bids	52
4.6	Model phase 4: Close auction	55
4.7	Model phase 5: Ready for deletion	61
5.1	Ganache IDE: displaying the user accounts, their balance and their tx count	66
5.2	Truffle migrate console results	67
5.3	All auction controller tests passed	73

Code Fragments

4.1	Auction controller state variables	40
4.2	Auction helpers	41
4.3	Auction state variables	43
4.4	Auction controller: deploy a new auction	47
4.5	The constructor of the auction smart contract	48
4.6	Bid in hidden round	50
4.7	Close hidden round	51
4.8	Bid in open round; revealing bids	53
4.9	Close auction	54
4.10	Find the auction winner	56
4.11	Transfer back deposits	58
4.12	Transfer highest bid and remaining deposits to seller	59
4.13	Retrieve token	60
4.14	Auction controller: delete auction	62
4.15	Delete auction	63
5.1	Truffle configuration	66
5.2	Truffle migration: deploy controller contract	66
5.3	AC test: architecture	68
5.4	AC test: can deploy new auction	69
5.5	AC test: can not delete if not admin	70
5.6	AC test: cannot delete auction prematurely	71
5.7	AC test: admin can delete auction	72
5.8	Test Auction, a helper contract for testing	74
5.9	Auction test: architecture	75
5.10	Auction test: initialization	76
5.11	Auction test: can bid in hidden round	77

5.12 Auction test: cannot bid if deposit is too low	78
5.13 Auction test: cannot bid in hidden round after hidden bids deadline . . .	78
5.14 Auction test: close auction if no hidden bids were received	79
5.15 Auction test: can bid in open round	80
5.16 Auction test: cannot bid in open round if bidder did not participate in hidden round	81
5.17 Auction test: cannot bid in open round if the open bid does not match the hidden bid	81
5.18 Auction test: close open round successfully	82
5.19 Auction test: simulate bidding with different accounts	83
5.20 Auction test: finding the correct winner	84
5.21 Auction test: did not transfer deposit back to invalid bidder	85
5.22 Auction test: sent highest bid to seller, as well as any remaining deposits	86
5.23 Auction test: winner can retrieve token	86
5.24 Auction test: only the auction winner can retrieve the token	87
5.25 Auction test: token is not callable	88

Chapter 1

Introduction

This chapter presents a brief introduction to the main topics of this thesis: microgrids, smart energy trading, and blockchain technology. Subsequently, we discuss our motivation behind creating a solution for blockchain-based energy trading. This is followed by the thesis definition, which outlines our research objectives. We will then discuss related work that has been conducted in the area of interest that influenced our work and attempt to provide our contribution to the field. Finally, we present an outline of the thesis structure with a brief description of each chapter in the thesis.

1.1 Microgrids

As the world steadily tries to combat the danger of global warming, renewable energy sources have been a topic discussed widely in the last years. Several countries, such as Kenya [1], France [2], Haiti [3], and the United States [4] have all successfully developed microgrid systems in the hopes of providing an alternative power supply to their citizens. The US Department of Energy [5] defines a microgrid as *"a group of interconnected loads and distributed energy resources within clearly defined electrical boundaries that acts as a single controllable entity with respect to the grid. A microgrid can connect and disconnect from the grid to enable it to operate in both grid-connected or island mode."* It is a self-sufficient energy system that can either operate in connection with the regional energy providers (macrogrid) or disconnect from the macrogrid and rely on renewable energy

sources (RES) to supply a discrete area with energy, such as a hospital, a neighborhood or a college campus, for instance.

While connected to the macrogrid, microgrids serve as a sustainable and resilient energy supplier when the national power grid experiences outages in cases of natural catastrophes or if an electrical wire is simply brought down by a fallen tree. In island mode, the microgrid is dependent on local energy sources like solar panels or windmills. The producers of such energy are called *prosumers*, as they can also be *consumers* of said energy. Prosumers will share the excess energy they produce with the rest of the microgrid network.

1.2 Smart Energy Trading

Modern energy systems are constantly adopting new technology in order to improve efficiency and flexibility [6]. With a global shift towards environmentally friendly technology, there is a significant amount of innovation in this field that can increase the utilization of green energy. The emergence of microgrid technology has given consumers an insight into their energy consumption habits and local energy resources. With access to electric storage units that can store excess energy, it is natural that prosumers would want to trade such a surplus of energy with local consumers. While regional energy providers have had a monopoly on energy distribution, they have had control over the availability and market price.

Even though energy trading has been available, energy brokers have served as an intermediary between producers and consumers. Energy brokers lower the costs of regular households, but they do charge a fee or commission for their services. Providing energy trading between local producers and consumers decentralizes the trading by removing the need for such intermediaries and the regional providers altogether, thereby minimizing the costs while also reducing the regional grid load. These trading systems are often implemented with auction mechanisms, where a prosumer acts as a seller and the consumers as bidders. The prosumers and consumers will then together dictate the market price.

1.3 Blockchain Technology

Blockchain technology was first introduced by Satoshi Nakamoto, an anonymous person or group of persons, in 2008 [7]. His whitepaper on Bitcoin was a revolutionary step in Information and Communication Technology (ICT). A blockchain is a digital public ledger where lists of *blocks* are cryptographically linked. Each block contains several transactions, its current timestamp, and a cryptographic hash pointer to the previous block in the chain. The blockchain is distributed to a peer-to-peer (P2P) network, where each peer is called a *node*. Each node manages its own local copy of the public ledger. The decentralized nature of the blockchain makes it immutable, as modifying one block would make it apparent to all the other nodes in the network that the block had been tampered with. As modifying a block would also modify its hash, the modified block would not be accurate according to the other nodes' ledgers.

This technology has been implemented in many other fields besides cryptocurrency, and one of these fields is energy trading. The transparency, immutability, decentralization, and security of blockchain technology are significant advantages that can improve the current energy trading solutions. Using smart contracts, which are computer programs that are transacted to the blockchain, trading systems can be automated and executed without the need of trusted third parties (TTP). Once deployed to the chain, the contract cannot be updated. As all transactions are transparent, trading participants can inspect the contract and all transactions associated with it.

1.4 Motivation for Blockchain-based Energy Trading

Utilizing blockchain technology for energy trading leads to a decentralized network where the market is controlled by the consumers rather than centralized regional providers. It can make the market competitive and react to supply and demand rather than forcing consumers to pay extra fees to energy brokers and energy providers that have a monopoly on the energy market. Moreover, trading energy in microgrids will lead to a growth in renewable energy usage. Introducing blockchain technology into the solution will also make energy trading more efficient as it is autonomous. However, creating such a system leads to problems with bidding privacy due to the blockchain's inherent transparency. Bids cannot be placed in the blockchain in plaintext as every transaction is transparent in the network logs. Nevertheless, bidders must transfer *something* to the auction in

order to avoid fraudulent attempts to win the auctioned off energy with a high bid that the bidder does not actually possess. This dilemma creates an interesting problem that must be solved in order to create a successful blockchain-based energy trading platform.

1.5 Thesis Definition

As discussed in the previous section, a transparent, efficient, and immutable energy trading platform will lead to lower consumer costs and higher prosumer profits while reducing the energy footprint. Providing a blockchain-based eAuction system to prosumers and consumers will eliminate the need for energy brokers and intermediaries and gives control of the energy market back to the prosumers and consumers. This thesis will focus on the problem of designing a secure, blind auction system on a transparent blockchain network. Moreover, the designed solution model will also be implemented as a functioning prototype where technical details of coding the system will be demonstrated. It is crucial to apply security by design in both the solution model and the code in order to meet the user's security requirements in a system that tackles transactions of Ether.

1.5.1 Research Objectives

The following research objectives must be realized in order to fulfill the thesis definition:

- RO 1)** Model an appropriate, secure blockchain-based eAuction solution that is capable of fulfilling the abovementioned requirements.
- RO 2)** Implement a secure prototype of the auction model.
- RO 3)** Evaluate and analyze the prototype.

1.5.2 Research Questions

In order to satisfy our research objectives, it is necessary to answer the following research questions.

- RQ 1)** How can we ensure transparency while hiding the auction bids?
- RQ 2)** How can we validate that the bidders do indeed have the funds they are offering when they cannot transfer the bid itself?

RQ 3) How can we evaluate the functionality of the implemented solution?

RQ 4) How can we assess the security of the solution?

1.5.3 Scope

The thesis focuses on what technology is most relevant in the creation process of the platform. We will focus mainly on designing a solution model of the energy trading platform and its code implementation. It is important to note that the actual transfer of energy from a prosumer to a consumer is considered out of scope. How the auction winner will retrieve the energy provided by the seller is a conceptual idea only, and will not be considered further. We have also made some basic assumptions that enable the thesis to maintain focus on the aspects that contribute to the field. The main assumptions of the thesis are:

1. The system's user interface (UI) is the interface that users use to interact with the solution. This interface is only conceptual and is assumed to be securely coded.
2. The connection between users and the UI is assumed secure. This assumption ensures that we can focus on our blockchain-based eAuction solution and not shift the focus to human errors and technical attacks that target credential UI pages, input validation from the UI, and similar attack surfaces.
3. The actual transfer of energy from the seller to the auction winner is considered out of scope; we will focus on the eAuction system and the Ether payments on the blockchain.

1.6 Related Work

This section describes research that has been conducted in our area of interest related to this thesis. The related research has organized into two main topics: smart grids and blockchain based trading systems.

1.6.1 Smart Grids

Microgrids have traditionally been used in the power sector in order to support existing grids in events like power failure or to provide another form of energy production if the

grid's primary production is unstable [8]. The ability to operate as a separate grid while still being connected to an external grid has allowed for several solutions that promote a more decentralized grid.

"The Future of the Electric Grid" which is an interdisciplinary study from Massachusetts Institute of Technology, states that *"the best way to describe a smart grid is the expanded use of new communications, sensing and control systems throughout all levels of the electricity grid."* [6] With a global shift towards environmentally friendly technology, there is a significant amount of innovation in this field that can increase the utilization of green energy. Some smart grids have implemented energy production into their systems, often using solar panels to both produce and consume energy, becoming a prosumer. In a system where some users are prosumers, there should exist trading platforms that would allow the users to sell their excess energy if they produce more than they consume.

The International Renewable Energy Agency (IRENA) is an organization that promotes possible solutions that can lead to a more sustainable energy future. In 2019 they published "Blockchain – Innovation Landscape Brief". The brief is a part of their project to illustrate the need for synergy between technologies to create solutions to renewable and sustainable energy problems. The brief promoted blockchain technology as some of the most promising technology to solve the immediate future's energy problems [9].

1.6.2 Blockchain-based Trading Systems

Blockchain has been commended for its transparency and integrity without the need for a trusted third party. The decentralized nature of such a system makes them ideal for some applications [10]. One such application is auctioning. eAuctions created a system for buyers and sellers to trade over the internet, removing the need for physical presence. The blockchain provides transparency and integrity in a mathematically proven way. However, other problems may occur based on the desired auction type [10]. The transparency may mitigate corruption, which has been an issue in countries like Ukraine. However, the auction's input data should remain confidential in some auctions to provide a suitable auction type where the bidders have a mutual distrust, which prompts them to desire that their bid remains confidential towards competitors. Auctions refer to this as sealed bids. A sealed bid auction keeps the bids confidential until the deadline, where the bids are revealed [11].

The Ethereum blockchain offers new possibilities for trading systems [12] using blockchain-enabled decentralized applications and smart contracts. Smart contracts enable more flexible agreements to be made and a significant reduction of complexity to both create and use decentralized applications. An auction hosted on a system like Ethereum effectively merges the auction mechanism in the application with the blockchain attributes [13] that is provided by the protocol network. The smart contract determines the auction winner in a verifiable manner by other users instead of a trusted auctioneer acting as a third party. In addition to the decentralized nature of such a system, an auction that performs the transactions in the auction On-Chain will utilize an infrastructure that provides more predictability in service fees, which can be beneficial when designing, and later using the system.

The energy sector has in recent years identified the potential of blockchain-based energy trading systems [9]. Blockchain-based assets are described as being a promising method to raise capital for projects, meaning that some projects can receive funding without the need of a larger actor. This results in the potential of more crowdfunded projects, further promoting the decentralization of these systems. The authors of [13] proposed a double auction mechanism to facilitate peer-to-peer energy trading. [9] states that the decentralized model based on blockchain has the potential to bring down prices through increased competition and grid efficiency. A more decentralized power system operation could overall lead to better utilization of grid assets, improved green energy utilization, and allow the consumers to have more influence in the energy market by reducing some of the current influence that energy brokers possess.

1.7 Contribution

We solved the problem stated in the thesis definition by meticulously designing a secure eAuction model utilizing the Ethereum blockchain network. This solution model was implemented by writing smart contracts in Solidity and deployed to Ethereum's test network Ganache, where thorough unit tests were performed. The security of our solution has also been analyzed with respect to the user's security requirements. Although similar eAuction solutions exist already, most of them propose a possible solution model of a blockchain-based energy trading platform. This thesis also provides a working prototype of the auction. This is a contribution to the field of emerging Decentralized Applications (DApps) on Ethereum. Using a network like Ethereum that has been called the second

generation of blockchain, both provide an abstraction of a significant amount of mechanisms but also facilitates applications of different complexity and scope than systems based on Bitcoin and older generations of blockchains.

Other similar works have either had the structure of a whitepaper, proposed a solution of a new blockchain protocol, or addressed another unrelated problem where their solutions consequently have different functionalities than our proposed solution. Our solution could, without a significant amount of work, be altered to facilitate the trade of other objects like Non-Fungible Tokens (NFTs) or other cryptocurrencies, as it also includes a working minimal viable product (MVP). Moreover, the process could be repeated to create another system with similar functionalities. If the Proof of Stake (PoS) system is implemented in Eth 2.0, this solution can be implemented on that new network, or the logic of the code could also be migrated to another similar system if that would be more effective.

1.8 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2, Theoretical Background: In this chapter, we have discussed the most relevant theory required to understand the solution. The chapter starts with an introduction to the history and functionality of blockchain. Subsequently, the Ethereum network is described. This section focuses on the ecosystem of Ethereum, which introduces the most important features and ideas behind our trading system. Finally, we discuss smart contracts on Ethereum, both their evolution and their current state, and introduce relevant development tools and languages for writing smart contracts in Ethereum.

Chapter 3, Blockchain-based eAuction Solution: This chapter introduces our proposed solution. It introduces the case that is the foundation of our thesis, the desired functionality of a trading system, and the relevant issues within such a system. The solution is then divided into five phases of an auction life cycle which is described and illustrated using sequence diagrams, focusing on how the users interact with the platform.

Chapter 4, Implementation: This chapter is dedicated to the implementation of our solution. It introduces the platform architecture and the system components before demonstrating how the solution is implemented with the help of detailed code fragments.

Chapter 5, Implementation Assessment: In this chapter, we discuss how the implementation has been tested and analyzed to ensure platform integrity and functionality. Furthermore, a security analysis of our solution has been conducted that provides insights into how users' security requirements have been met.

Chapter 6, Conclusion: This chapter will summarize the main accomplishments of our thesis, and discuss the research objectives and research questions with respect to the work. Additionally, we include possible improvements that can be integrated into the solution as a part of future work.

Appendix A: The complete source code of the Auction Controller contract, written in Solidity.

Appendix B: The complete source code of the Auction contract, written in Solidity.

Appendix C: The complete source code for the Auction Controller tests, written in JavaScript.

Appendix D: The complete source code for the Auction tests, written in JavaScript.

Chapter 2

Theoretical Background

This chapter focuses on introducing the theoretical concepts that provide the required knowledge to understand the underlying technology that is needed both to develop and implement the energy trading platform. First, we describe blockchain technology and its features in detail. Then, we discuss the main topics of Ethereum and its ecosystem while maintaining focus on the scope of the thesis. Moreover, the cryptographic primitives applied in the solution will be described. Finally, we present the core development tools used to implement the solution.

2.1 Blockchain Technology

This thesis is an application of blockchain technology. This subsection provides some background information and history of blockchain technology and introduces some core concepts relevant to the subsequent chapters.

2.1.1 Blockchain History

The idea of a decentralized digital currency system has been around since the 80s with concepts as Chaumian binding, which proposed a method of anonymous payment using blind signatures [14]. In the late 90s to mid-2000s, concepts like Wei Dai's "b-money" [15] and Hal Finney's "reusable proofs of work" [16] were introduced. However, most of these concepts were theoretical and struggled with the decentralization of the systems, and therefore relied on a trusted computing backend. The Bitcoin whitepaper "Bitcoin:

A "Peer-to-Peer Electronic Cash System" was published in 2008. The paper introduces a solution that establishes ownership through public-key cryptography with a consensus algorithm. This algorithm is known as "proof of work". As a consensus algorithm, it allows nodes to collectively agree on updates to the states of the Bitcoin ledger. Bitcoin removed the need for trusted third party computation, and instead, the legitimacy of transactions was proven cryptographically. The computational power came from participants that used peer-to-peer connections to run the Bitcoin network. Bitcoin is today known as the first practical implementation of a distributed ledger system.

2.1.2 Blockchain networks

In order to design and develop blockchain applications, a basic introduction of some of the core concepts in blockchains is explained in the following subsection.

Blocks are all linked to the previous block in the chain, and every block contains some transactions. Their hash value can identify every block in the chain. Every block also includes the previous block's hash, in what is called the block header (illustrated in Figure 2.1). This chain of linked blocks is connected all the way back to the first block that was created on the chain. This block is known as the genesis block. Due to the previous block's hash being included in every block, an attempt to change that previous block would also affect the current block hash, leading every child's block hash to change. This mechanism ensures that if someone tries to tamper with a block, every subsequent block will also be changed, leading to a whole different hash in every block. This mechanism ensures that the blockchain's history is immutable.

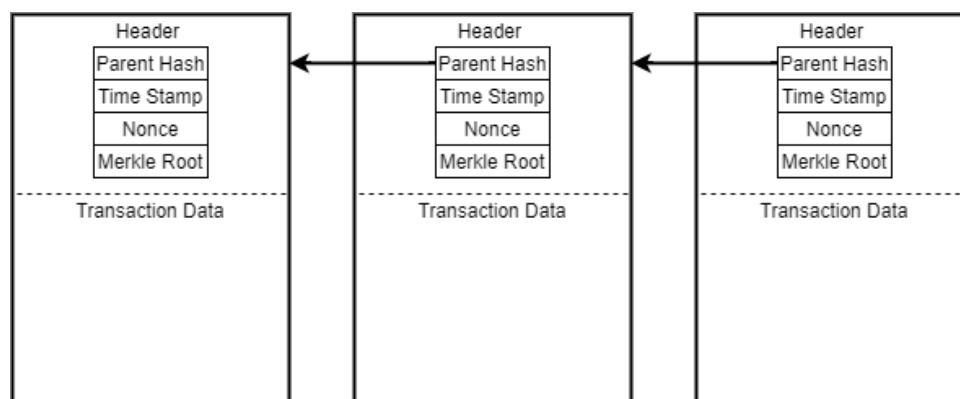


Figure 2.1: Simplified blockchain

Transactions is the process where a blockchain account transfer a certain amount that blockchain's currency to another owner. The owner adds a signed hash of the previous transaction and adds the next owner's public key at the end. This method ensures that anyone can verify the chain of ownership (see Figure 2.2).

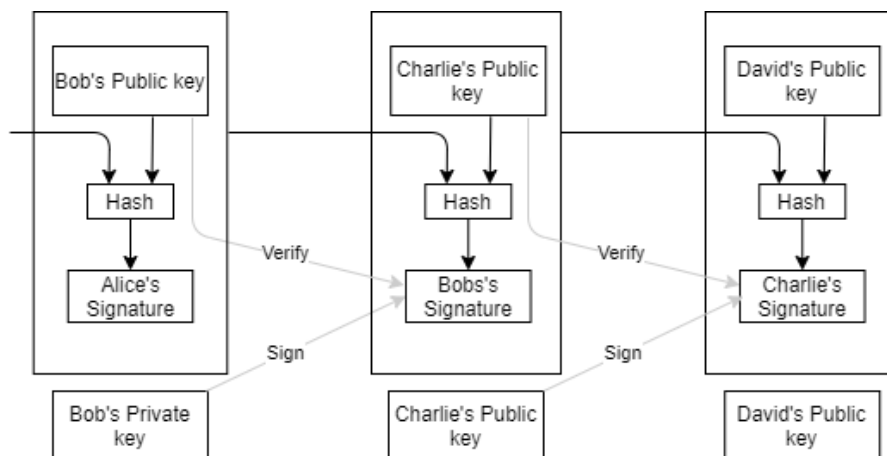


Figure 2.2: Transactions in blockchain

Node: Part of the blockchain network, a full node is a participant that stores a complete copy of the blockchain. Similar to all participants, a node can leave and reenter the network at their own will. The cooperation of nodes is a crucial part of the network.

Consensus: All transactions are publicly announced. The nodes need to agree on a common history of all these transactions. The consensus protocol is how the blockchain system takes these types of decisions that need a majority of the nodes to agree.

Proof of Work: A cryptographic zero-knowledge proof that is used to incentivizes the participants in the network. PoW is a consensus algorithm that ensures that the miners have an incentive to secure the blockchain in the form of currency or a portion of the payment from fees on the network. The deterrent for dishonest miners comes from the monetary loss from the energy cost that is spent attempting to mine a fake block. The most popular blockchains use this consensus algorithm.

Proof of Stake was originally a precursor of the PoW, based on financial stake. PoS revolves around that owner of the blockchain's base cryptocurrency can lock up their currency into a deposit, becoming a validator. The validators purpose and vote for the next valid block in the block creation. Each validator's voting power correlates to the value of the deposit. This is called *staking*, and validators are incentivized to perform

staking to receive a reward in the currency. The difference in punishment is that the PoS system punishes the dishonest staker by a loss of their staked amount of currency.

2.1.3 Features of Blockchain

Our thesis revolves around blockchain technology. In order to understand some of the core functionality of blockchain, this subsection introduces some of the features of blockchain systems.

1. Decentralization

A decentralized system like blockchain base the security of the network on proof of work, instead of traditional access control, and allow for full transparency. In comparison, a centralized system, like a traditional payment network or a bank, relies on access control schemes to allow certain entities access to resources and make decisions. The decentralized model ensures that the user retains more of the power. There will be no banks that have access to private accounts in the form of private keys. However, if the user is the victim of a hack, there is no third party able to revert transactions.

2. Transparency

This decentralized consensus allows all the nodes in the network to agree on a common record of ownership. This common record is transparent. Transparency is accomplished due to the fact that all the transactions are accessible by searching the blockchain; all the transactions and corresponding data between the public addresses are available. This transparency comes from the design of the network. Instead of a centralized entity giving "transparency" by providing data, transparency is built into the system and eliminates potential bias. Regarding cybersecurity, transparency provides what has lately been an issue with large companies regarding security disclosures of events. If an attack was made towards a blockchain, the attack details would be visible and public during the attack instead of presented to users at a later date.

3. Immutability

Immutability in the blockchain refers to how the data is unchangeable once it has been posted to the blockchain. Since all records are based on the consensus of the nodes, every node stores a local copy of the ledger. Once a block is successfully attached to the ledger, no node can go back and change any property of that block. As mentioned in the previous subsection, if someone tried to alter data on the block, this would result in a

different hash than what the other blocks have stored. This change would also result in every preceding block changing its hash, requiring a substantial amount of computational power, especially in longer blockchains.

2.2 Ethereum

Vitalik Buterin published the Ethereum Whitepaper in 2013, and the Ethereum network has been active since 2015. This section will describe some of the core concepts of Ethereum, such as the ideas in the original whitepaper, the ecosystem of Ethereum, and how smart contracts function on the Ethereum network.[17]

2.2.1 Ethereum Whitepaper and Core Ideas

The main idea of the original whitepaper of Ethereum was to propose an alternative blockchain protocol that would be better suited than Bitcoin to build decentralized applications. The goal was a blockchain protocol designed for the primary purpose of building applications instead of functioning as a decentralized currency. Faster development time, security, and enabling of application interaction were key features proposed in the whitepaper. In order to achieve this goal, the paper proposed that the best solution would be "the ultimate abstract foundation layer"; a built-in Turing-complete programming language in the blockchain. This language is known today as Solidity. The philosophy behind the design of Ethereum is stated in the whitepaper to promote the following principles [17]:

Simplicity: The protocol should ideally be simplified so that the average programmer would be able to participate. At the center of the protocol is the vision that Ethereum is a protocol that is open to all. Any addition of complexity should provide a substantial benefit or be discarded.

Universality: Ethereum does not have "features". Ethereum provides the internal scripting language, which a programmer can use to construct mathematically defined smart contracts or transactions. These contracts can interlock and provides more possibilities; ultimately, any functionality should be programmable using the Ethereum protocol.

Modularity: In order to benefit the entire cryptocurrency ecosystem, modularity is desired. Modularity would allow changes to apply each of their separate modules, and

the application stack would continue to function without any modification. Then some of the Ethereum features would be accessible in other protocols, even if their features are not required in Ethereum.

Agility: The Ethereum protocol is flexible. Proposed changes that change the architecture or high-level constructs will be under heavy scrutiny, but if an opportunity to promote the core ideas or solve the main problems is found, it may be implemented.

Non-discrimination and non-censorship: All regulatory mechanisms should focus on security risks and other direct threats, not stopping specific applications.

2.2.2 Ecosystem of Ethereum

In order to get a working grasp of the ecosystem of the Ethereum network, this subsection introduces some of the features that are most important when using the Ethereum network as a developer and a user.

Ethereum accounts are objects in the Ethereum protocol that are designed to represent a typical account function. There are two types of Ethereum accounts: an externally owned account or a contract account. The externally owned accounts are accounts that the private key owner controls, while the contract account is controlled by its smart contract code. The Ethereum account is derived from the last 20 bytes of the Keccak-256 hash of the public key[18]. The externally owned accounts are usually managed by an Ethereum wallet.

An **Ethereum wallet** is a software application that stores the relevant cryptographic keys and broadcasts transactions for the user. The wallet functions as the primary user interface to Ethereum for most users. A wallet manages the keys and addresses, controls access to the cryptocurrency, tracks the balance, and manages transactions. A wallet abstracts some of the functionality that is considered best practice when utilizing cryptocurrencies. One of them is the managing of several private keys. Due to pseudo-anonymity, most Ethereum wallets manage several private keys while using new keys for some transactions. The wallet itself does not hold any currency. They just store the keys, which then the user can authorize the wallets to sign the transaction, which then sends the currency. There are several popular wallets. For this thesis, MetaMask is used.

Ether is the "currency" of Ethereum. It is used as a currency to trade with and to pay for the transaction fees. An entity needs an ether balance to pay for computation on the Ethereum network.

Transaction in Ethereum refers to the signed data package that contains a message from an externally owned account. The signed data package is transmitted by the Ethereum network, and the transaction is recorded and stored on the Ethereum blockchain. These transactions contain, like most cryptocurrencies, the recipient of the data, a signature from the sender, and the amount of currency sent. In addition, Ethereum transactions contain an optional data field, **STARTGAS**, which describes the maximum number of computational steps the transaction execution is allowed to take, and **GASPRICE**, which is the fee for each computational step.

The latter two are a part of Ethereum's service model, which attempts to prevent and reduce unwanted computational wastage, both accidental or malicious. **Gas** is a unit that is used as a unit to represent the cost of computation [19]. The gas costs are calculated based on several factors like computational steps and byte length of the transaction. The gas system is intended to make users of the network pay fees proportionately to the number of resources they use, which allows for the incentivization of miners and increases costs for potential attackers.

2.2.3 Smart Contracts

The term "Smart Contracts" was coined by Nick Szabo in the 1990s [20]. In his paper from 1994, he states that "*A smart contract is a computerized transaction protocol that executes the terms of a contract.*". The smart contract should be designed to satisfy the common contractual conditions while minimizing the exceptions. These exceptions could both be accidental or malicious, meaning that the smart contract should align with economic goals like lowering fraud cost, enforcement cost, and other transaction costs. In later work, Szabo referred to vending machines as a concept that could be considered a primitive physical ancestor of smart contracts [21]. In a vending machine, the logic is preprogrammed, ensuring that a specific input like currency and a choice will result in a corresponding output. This metaphor has in later years gained popularity and is often used to give an idea of what a smart contract is.

Smart contracts in the modern-day most often refer to a similar concept that is in some way or form deployed using blockchain. Smart contracts in Ethereum is a term used to refer to immutable computer programs that run on the Ethereum Virtual Machines as a part of the Ethereum network. The smart contracts are deployed to the blockchain, making them *immutable*. There is no way to modify a deployed smart contract if developers want to modify the contract, they need to deploy a new instance. The contracts are *deterministic* meaning that the outcome with a particular input will be the same for all users, every time given the same context.

2.3 Cryptographic Primitives in the Platform

This section introduces the cryptographic primitives that are utilized on the Ethereum network. These primitives will be explained in a high-level context to give adequate background for the coming chapters.

2.3.1 Hash Functions

Ethereum utilizes cryptographic hash functions extensively through the network. Hash functions transform Ethereum public keys into public addresses, which all users use. The functions are also used in data verification. *A cryptographic hash function maps data of arbitrary size to a fixed size string of bits.* The function only works one way, which means that the only mathematical way to recreate the output hash is with the same input. If the hash function is cryptographically secure, the only way to determine a possible input for a given hash output is to brute-force inputs into the function, searching after the matching output. However, the possible input does not have to be the original input due to the hash function's "many-to-one" nature. This is called a hash collision, and good hashing algorithms should make hash collisions near infeasible to find. In Ethereum, hash collisions are so rare that it is practically infeasible [22]. Hashing algorithms are utilized by many security applications like data fingerprinting, proof of work, message commitments, and authentication. At the current state of Ethereum in June 2021, Ethereum uses the **Keccak-256** hashing algorithm primarily [23].

2.3.2 Digital signatures

Digital signatures are used to sign messages and ensure integrity and authenticity. In Ethereum, the digital signature is created by using the Elliptic Curve Digital Signature

Algorithm (ECDSA) to combine the message in the transaction with the private key. The digital signatures are used for transactions in the Ethereum network when an account wants to transfer Ether. A digital signature corresponding to the Ethereum account must be created using the private key and send with the transaction. Elliptic Curve is used in signatures because any party can verify the signatures. The signature creation requires the private key of the sender's Ethereum account. However, the verification process does not. The verification process is used to confirm that the signature in the transaction corresponds to the private key that is linked with the public key of the sender's Ethereum address [24].

2.4 Blockchain Development

This section will describe Smart Contracts from an Ethereum development perspective. Smart contracts consist of compiled code that is deployed to the blockchain; once transacted to the blockchain, a contract cannot be updated because of the blockchain's inherent immutability. Smart contracts can store arbitrary states and execute arbitrary functions when end users interact with them. Users can use transactions to interact with a contract. Ethereum is the most popular blockchain platform for the development of smart contracts [23]. Anyone can write and deploy a smart contract to the Ethereum network. The only requirements needed to deploy a contract are coding knowledge and the gas fee, which is negligible.

2.4.1 Solidity

As mentioned, Solidity is the Turing-complete programming language that is designed to create an abstraction of the foundation layer of Ethereum. Influenced by languages like JavaScript, Python, and C++, Solidity is a high-level language oriented around implementing smart contracts [25]. At the current time, Solidity is the primary language on Ethereum [26]. This subsection will describe several main aspects of the Solidity language that will be used in the implementation in Chapter 4.

State variables

State variables are permanently stored on the blockchain in the contract's storage and are declared in the global scope of the contract. Variables must be declared with their

intended variable type, as the necessary space will be allocated on the blockchain at contract creation.

Types

Each variable that is not instantiated with a value will be instantiated with the variable type's default value, as `null` and `undefined` does not exist in Solidity. The crucial types used in the implementation and their default values are listed below.

- `bool`: `false`
- `uint`: `0`
- `address`: `0x00`
- `array`: `[]`, empty array
- `string`: `""`, empty string

The `address` type is special to the Solidity language. It holds an Ethereum account address; either an externally owned account (a user account address) or a contract account address. The address can also be declared as `payable` which gives the address two extra members: `transfer` and `send`. This marks the address as an address that can receive Ether from the contract.

A mapping type in Solidity refers to a variable with the syntax `mapping(KeyType => ValueType) mappingName`. Mappings are initialized in the same way as regular hash tables, where it is assumed that *"every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's default value"* [27]. Mappings do therefore not have a member `length`, and a single key cannot be repeated in the same mapping. To access the value of a mapping, you have to access it via its key.

Another important Solidity type used in the implementation is `struct`. Structs are customizable groups of variables and are tightly packed in the EVM to reduce the storage space it obtains. Grouping associated variables will therefore reduce the gas price of storing new variables on the blockchain.

The `enum` type is used to create a finite group of constant values, which is often used to define possible states of a state variable or the contract itself.

Globally available types in Solidity are used to provide information about the blockchain's state. There are also globally available utility types. The two most crucial of these types are described below.

- **block**: This variable contains information about the blockchain and the current block. The one member that is used in the implementation is `block.timestamp`, which returns the *"current block timestamp as seconds since unix epoch"* [28]. This timestamp cannot be guaranteed to be completely accurate, as each block in the Ethereum blockchain takes about 13-14 minutes to generate. As the Solidity docs state: *"the only guarantee is that [the current timestamp] will be somewhere between the timestamps of two consecutive blocks in the canonical chain."* [28]
- **msg**: This variable provides information about a transaction, such as the sender Ethereum account address (`msg.sender`) and the Ether transferred (`msg.value`).

Function types

Function types declare the type of a particular functions. The *internal* type is assigned to functions that can only be called from inside the current contract. All functions are defaulted to internal unless otherwise specified (with a *public* keyword, for instance). *External* functions are the opposite: they can only be called from external entities. Functions can also be marked with the following three function types:

- **view**: Functions that do not modify state variables can be marked as **view** functions.
- **pure**: These functions are even more restrictive than view functions, as they can neither write or read state variables.
- **payable**: Functions that contain Ether transactions must be marked as payable. Without this keyword, the function will reject Ether transactions.

Reference types

Reference types are used to refer to values that can change size, such as structs, arrays, and mappings. When using these values, one must explicitly declare the storage type to be used. There are three different data storage locations in Solidity: memory, storage, and calldata. Storage is the most expensive in terms of gas, then memory, and finally calldata is the cheapest.

- **memory**: Values that are stored in memory lives in a function scope. The value only lives until the code in a specific function has executed.
- **storage**: Values stored in storage lives until the contract is deleted.

- `calldata`: This data location is allocated for the arguments of a function.

Error handling

While Solidity offers several methods of handling errors, our implementation uses only one: *require*. A `require` statement is used to check for conditions and reverts the transaction if the condition returns false. It is used to validate arguments of a function, the sender of a transaction, or the transaction value. It is possible to return a string that describes the error if the condition in the statement is not met.

Modifiers

Function modifiers are convenience inheritable properties for repeated `require` statements. Functions that use modifiers will check the condition declared in said modifier before executing the rest of the function. Modifiers are declared in the global scope of the contract.

Events

Events in Solidity are used as a logging functionality within a contract. Applications such as user interfaces (UIs) can listen to emitted events through an Ethereum client's RPC interface, for instance, Javascript's *web3* library. The logs are stored in the transaction receipt, which is available on the blockchain.

2.4.2 Truffle

Truffle is a framework designed to make development easier on Ethereum. It is a development environment that can be used for testing and as an asset pipeline through JavaScript. Truffle can be used to compile and deploy contracts to Ethereum's main network or one of its test networks. By using the Command Line Interface (CLI), a Truffle project can be easily initialized by running the command `truffle init`. That command sets up default contracts, test files, and configuration files within the current directory while also providing a local development blockchain server.

`truffle compile` will compile all contracts within the `contracts` folder in the project. If there are any compile-time errors in the code, an error will be thrown and logged to the console. `truffle migrate` will run the migration scripts within the `migrations` folder. These scripts are where the contracts are deployed to the network. `truffle test` is used to run the test files in the `test` folder. Truffle tests can be

written in both Solidity and JavaScript. Note that examples of the migration scripts and truffle tests are demonstrated in Chapter 5.

2.4.3 Ganache

Ganache is a personal blockchain used for Decentralized Application (DApp) development, and is a part of the Truffle Suite. A personal (private) blockchain is a blockchain network where one person has control of the blockchain network, which is apt for a development environment. Ganache can be used as a CLI tool, but also provides a user-friendly UI where an end-user can view all transactions, generated blocks, and logs emitted from the development blockchain. Figure 2.3 illustrates an example of the UI. Ganache also provides ten accounts which are all equipped with 100 Ether each. It is important to note that this is not real Ether, and cannot be used in the main network. When configuring Truffle to use the Ganache network, these accounts can be used in the Truffle tests, and the balances will be updated in real-time. Ganache can also be used to inspect the contracts linked in the Truffle configuration.

The screenshot shows the Ganache UI interface. At the top, there are navigation tabs for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS. Below these are various status indicators: CURRENT BLOCK (4544), GAS PRICE (2000000000), GAS LIMIT (6721975), HARDFORK (MUIRGLACIER), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), MINING STATUS (AUTOMINING), and WORKSPACE (AUCTION). A search bar is located on the right side of the top bar.

The main content area displays the MNEMONIC (deposit nerve joke give gasp sibling flock lift claw riot morning slush) and the HD PATH (m/44'/60'/0'/0/account_index). Below this is a table of accounts with their addresses, balances, transaction counts, and indices.

ADDRESS	BALANCE	TX COUNT	INDEX
0xaF30872c7fa4A456f63a44f89240A6b09090fEfD	48.57 ETH	2540	0
0x034d599d1aaa6a9AaA47F8cCcaCE5F54eCc9E79B	98.68 ETH	684	1
0x91D29f811BE9D7be802EB0Dd9d0d3089D6d80C18	99.26 ETH	351	2
0xD8E3c73A91da6ED1b31A26350e878160B5c6f667	99.30 ETH	323	3

Figure 2.3: Ganache UI

Chapter 3

Blockchain-based eAuction Solution

This chapter presents the proposed blockchain-based auctioning system for smart energy trading in detail. It will focus on the functionality of the solution and how the technology is used. We start the chapter by outlining the case for which the solution is proposed. Then, we explicate our solution by dividing it into five phases: (1) Create an auction; (2) Transfer hashed bids; (3) Reveal bids; (4) Find the auction winner and perform closing logic; and (5) Delete the auction.

3.1 Case Description

Before we discuss the details of the proposed solution, it is quite important to introduce the case on which the solution is built around.

As it currently stands, regional providers of energy have a sole monopoly on the electrical grid to which all their citizens are connected to. Nevertheless, producing renewable energy through solar panels, small windmills, i.e., is growing more and more common. Microgrids are becoming a growing interest, where groups of people in near local vicinity can trade the excess energy they have produced. Being able to trade energy between different households without the need for an energy broker would allow the consumers to regulate the market price instead of the regional providers. A decentralized system

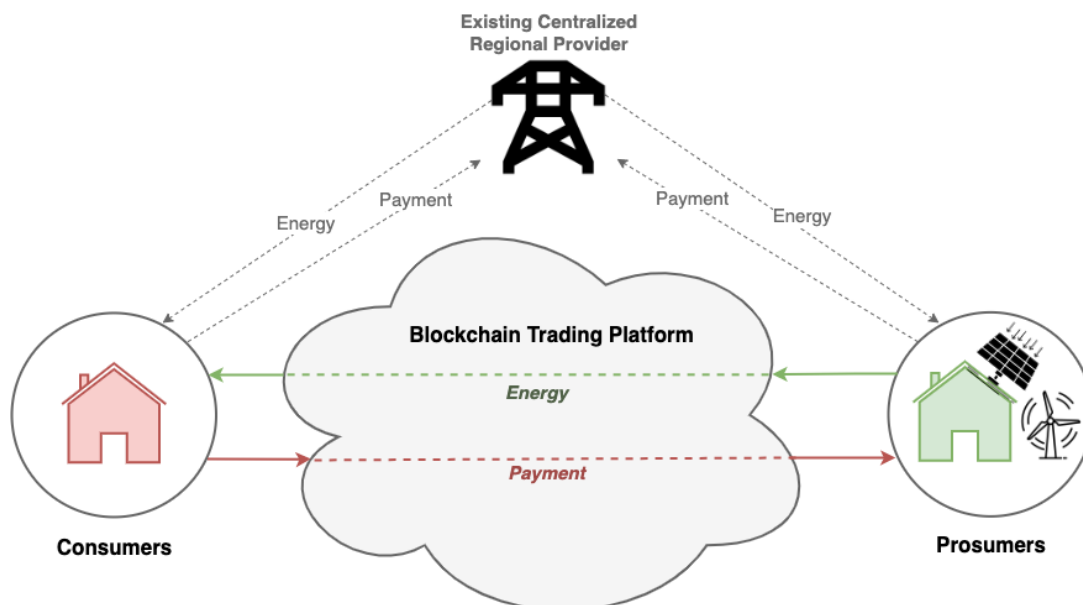


Figure 3.1: Overview of desired system

will reduce the power regional providers holds on electricity prices, lower costs of trading by removing energy brokers from the ecosystem altogether, and also increase the green energy utilization of local microgrids.

Cryptocurrency and the potential of Decentralized Applications (DApps) can be used to implement a decentralized trading platform (explicitly an auction system) in order to utilize the benefits of blockchain technology – transparency, automation, security, and efficiency. Such a system is illustrated in Figure 3.1. In a Decentralized Trading Application, users can trust that the auction they participate in will not be fraudulent as they can see each transaction that has been made in the blockchain logs.

There are two types of users in the network: (1) Prosumers, that produce more energy than they need and therefore want to sell their excess energy; and (2) Consumers, who want to buy green energy from those with a surplus instead of (or in addition) from their regional provider. A prosumer in this context is someone who is both a producer of energy and a consumer of energy. The prosumer can both buy and sell energy from/to other prosumers. The auction system should focus on facilitating energy trading that minimizes each user type’s time consumption and maximizes ease of use. In order to provide market stability and avoid bidding wars, the bidding process should be hidden. A problem then arises: if the bids should be hidden, but the blockchain is transparent, how would this be

feasible? Furthermore, each bidder must verify that they indeed do have the necessary funds for the energy transaction to ensure system integrity. The bidders must also trust that their funds are secure and that these funds will be returned to them at the end of the auction if they do not win. Moreover, the auction winner must receive a trusted token which serves as a verification of the energy they will acquire after the auction has closed, and this token should only be used once and be non-fungible. These requirements are the foundation of the solution which we will discuss throughout the remainder of this report. The physical aspects of connectivity and components that could facilitate energy trading are considered out of scope.

3.2 Solution Overview

Figure 3.2 illustrates the overview of our proposed solution. We have two different actors: the seller and the bidder(s). These interact with a Decentralized Application (DApp) which functions as our auction system. The DApp is located in the Ethereum network. The overview is split into five different phases, which are represented by different colors in the figure (the same color code is used throughout this report). The five phases comprise creating a new auction, two different bidding rounds, closing the auction and performing closing logic, and finally deleting the auction when it is finished. Each phase will be discussed independently from Subsections 3.2.1 to 3.2.5. Table 3.1 displays the overview of every abbreviation and notation that is used throughout this chapter.

Notation	Description
e	The energy amount that will be auctioned off by the seller.
bv	The minimum bid value that the seller selects when creating an auction. All bids that are lower than this value will be discarded.
dv	The deposit value that the seller selects when creating an auction. Every bidder must transfer at least this deposit amount when bidding in order to participate in the auction.
b_i, b_w	The bid corresponding to the i^{th} bidder (bidder i) and the bid of the auction winner respectively.
z_i	The salt used to hash b_i .
d_i, d_w	The deposit chosen by the i^{th} bidder and the auction winner respectively.
t^*	The token created for the auction winner.
Sk_S, Pk_S	Secret and public key of the seller.
Sk_i, Pk_i	Secret and public key of the corresponding i^{th} bidder.
PI_S	Ethereum account address of the seller, called PI for Pseudo Identity.
PI_i	Ethereum account address corresponding to the i^{th} bidder.
PI_w	Ethereum account address of the auction winner.
PI_A	Ethereum account address of the auction.
$DT_{PI_i}^{PI_A}$	Deposit Transaction from PI_i to PI_A .
$DRT_{PI_A}^{PI_i}$	Deposit Return Transaction from PI_A to the relevant user.
$PT_{PI_A}^{PI_S}$	Payment Transaction from PI_A to PI_S .
$ETC_{PI_A}^{PI_w}$	Energy Token Commission from PI_A to PI_w .

Table 3.1: Abbreviations and notations used in the solution chapter

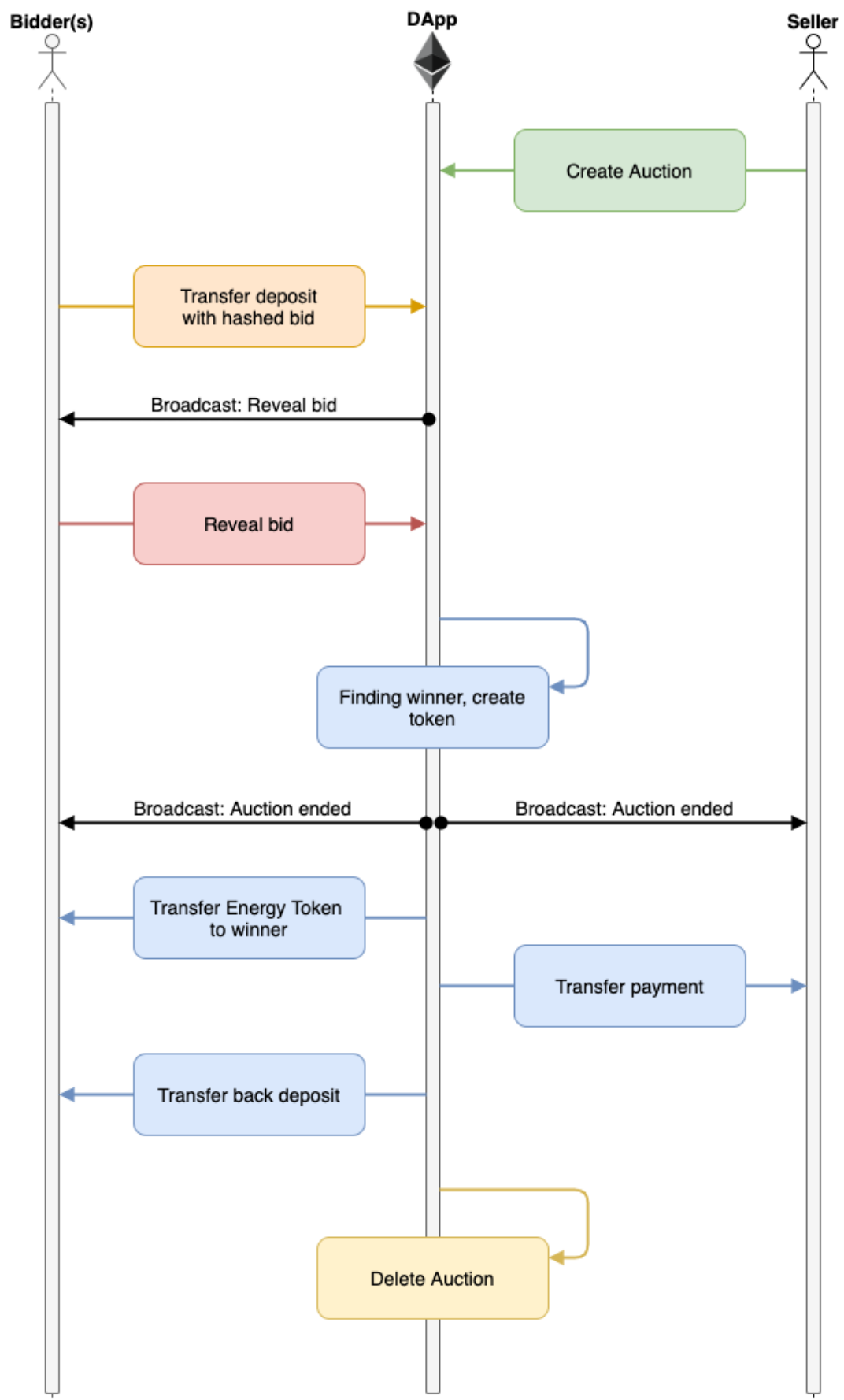


Figure 3.2: Overview of the proposed solution

3.2.1 Phase 1: Create Auction

Figure 3.3 presents the first stage in the auction process: the seller creates a new auction through the DApp. This is implicitly done by a four-way handshake. The seller accesses the platform and lets the DApp know about the user and its public key, and the DApp returns an acknowledgment that it has received the seller's access request. The seller can then send the necessary auction information with a *Create auction* request, which includes the amount of energy to be auctioned off (e), the minimum value for each bid (bv), and the deposit value (dv) each bidder must transfer with their hidden bid in order to be allowed to participate in the auction. This information is signed with the seller's private key (Sk_S) to ensure authenticity and integrity. As explained in Chapter 2, the user's Ethereum account is created with a cryptographic pair of keys: the user's public key (Pk) and secret key (Sk). When creating a new Ethereum account, the user will select a private key (can be a password, a passphrase, etc.). This key is then encrypted and used as the account's secret key. The user's public key is then generated from this secret key using the ECDS algorithm. The user's Ethereum account address (PI) is the last 20 bytes of the SHA3 hash of the user's public key. The DApp can therefore validate the seller by verifying the seller's signature with the help of the seller's public key, and will then create a new auction. Finally, it will send another acknowledgment to the seller to let the seller know that the auction was created successfully.

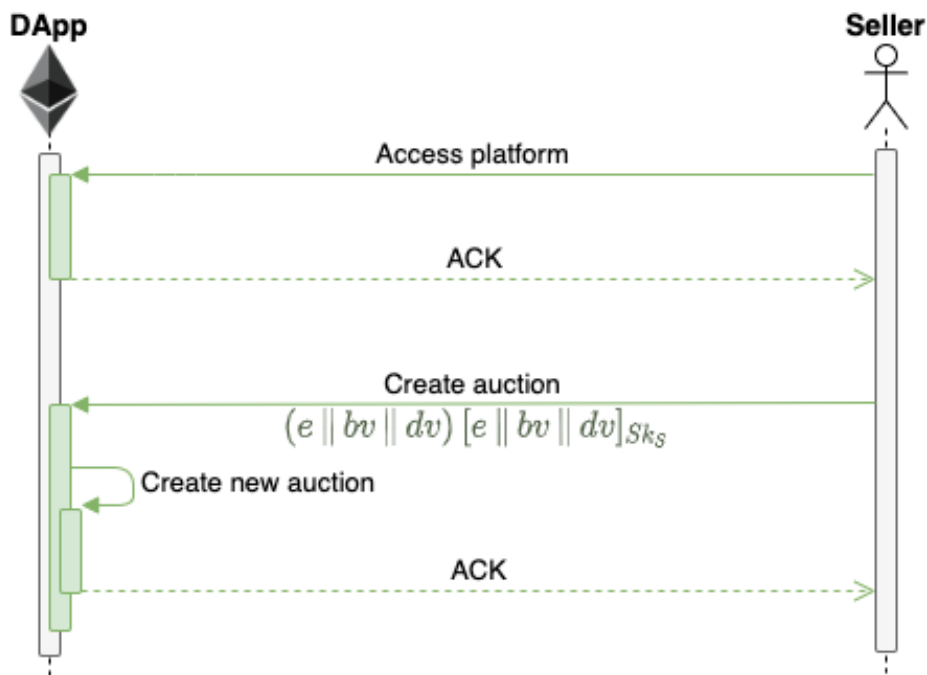


Figure 3.3: Seller creates auction

3.2.2 Phase 2: Hidden Round

During the next 24 hours after an auction has been created, it will be ready to receive hidden bids. The bid logic in this round is demonstrated in Figure 3.4 with respect to the i^{th} bidder in the auction. As in the previous phase, the bidder will request access to the platform and let the DApp register his public key. In return, the bidder will receive the auction information along with the auction's Ethereum account address ($e \parallel bv \parallel dv \parallel PI_A$).

The bidder will then post his bid by sending a transaction from his Ethereum account address (PI_i) to the auction's Ethereum account address (PI_A). Every transaction is denoted by its respective abbreviated name, a subscript representing who sent the transaction, and a superscript representing the entity that receives the transaction. Hence, this Deposit Transaction is denoted by: $DT_{PI_i}^{PI_A}(d_i \parallel H(b_i \parallel z_i))$.

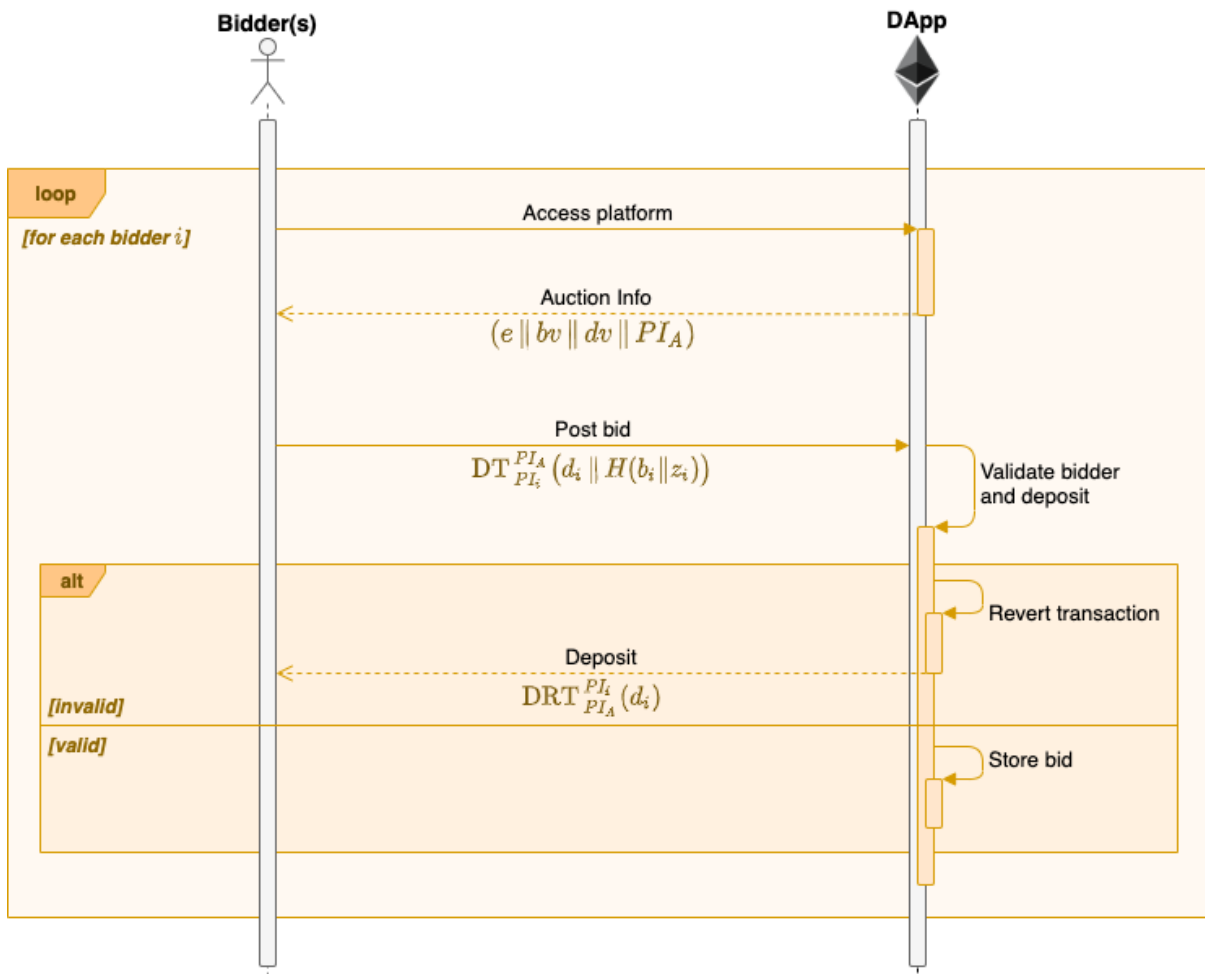


Figure 3.4: Bidder(s) transfer their deposit bid with their hashed bid

The transaction is sent from the Ethereum account address of bidder i (PI_i) to the Ethereum account address of the auction (PI_A). The transacted value is the bidder's deposit (d_i) paid in Eth, and the transaction message is the Keccak-256 hash of the bidder's bid (b_i) and salt value (z_i). Note that each transaction contains more information than what is noted here, but that information is omitted for brevity and clarity. The DApp will validate that the transaction was sent from the bidder that signed the transaction and that the deposit value transferred from the bidder is equal to or higher than the deposit value specified by the seller. If these requirements are not met, the transaction will be reverted, and the deposit will be returned to the bidder with a Deposit Return Transaction (to the i^{th} bidder's Ethereum account address (PI_i), from the auction's Ethereum account address (PI_A)). However, if all validation checks are passed, the bid will be stored on the blockchain along with the bidder's account address and the deposit value that was transferred.

This bidding round is designed in such a way that every bidder verifies that he has the necessary funds to bid on the auction without actually revealing the value of the bid. Note that deposits will be returned to the bidders in phase 4, except for the winner. The winner will receive his deposit subtracted by his bid. More information is given in phase 4 of the solution.

3.2.3 Phase 3: Open Round

When the hidden round is over, the auction advances to the open round, which will be open for another 24 hours. This is the phase where each bidder i will reveal the bid he submitted in the hidden round. Phase 3 is illustrated in Figure 3.5. The DApp initiates the round by sending a broadcast message to every bidder that participated in the auction, reminding them to reveal their bids. The bidder answers by sending the bid value and the salt used to hash the bid in the previous round in plaintext. These values are signed by the bidder's private key to ensure authenticity.

The DApp will first validate that the bidder is valid: that is, by checking that the incoming bidder's account address is one of the account addresses that were stored in the hidden round (in other words: this bidder participated in the hidden round). It is designed this way to ensure that users cannot hijack auctions in later rounds without transferring a deposit. If the bidder is deemed invalid, the bid will be discarded. However, if the bidder is verified, the DApp will investigate the bid. In order to validate the bid,

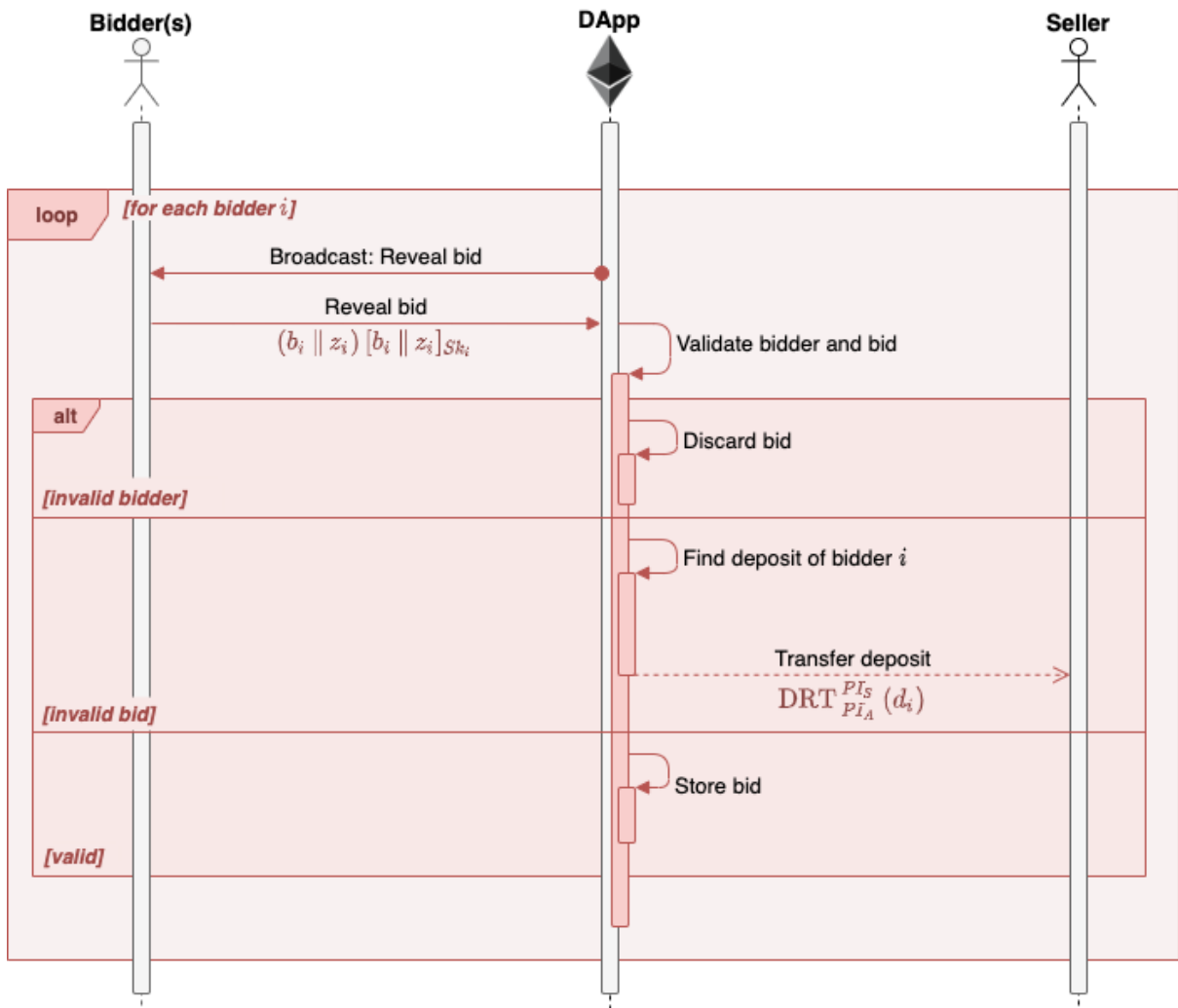


Figure 3.5: Bidder(s) reveal their bid

the auction will hash the plaintext bid with the salt given by the bidder by using the Keccak-256 hashing algorithm. It will then find the hashed bid that was stored along with the bidder's account address in the previous phase and compare the two hash strings. If they are equal, the bid is deemed to be valid. If not, the user has somehow tampered with his bid (i.e., tried to lower their bid), and his deposit will instead be transferred to the seller with a Deposit Return Transaction. This logic is implemented in order to deter bidders from fraudulent behavior. If both the bidder and bid are approved, the DApp stores the plaintext bid on the blockchain along with the hashed bid and deposit value belonging to the bidder's Ethereum account address.

3.2.4 Phase 4: Close Auction

When the two bidding rounds are over, the DApp will close the auction (presented in Figure 3.6). It first broadcasts a message to all users in the auction to let them know that the auction has ended and is no longer receiving bids. Next, it will loop through every stored open bid to find the highest value. The Ethereum account address associated with this bid is the auction winner. If there are several winner bids with the same value, the winner will be the bidder whose bid was first recorded on the blockchain. A token will then be created for the winner; it will contain the auction’s Ethereum account address (for traceability), the winner’s Ethereum account address, the energy amount the winner can retrieve, and the timestamp for how long it is valid (12 weeks from token creation)

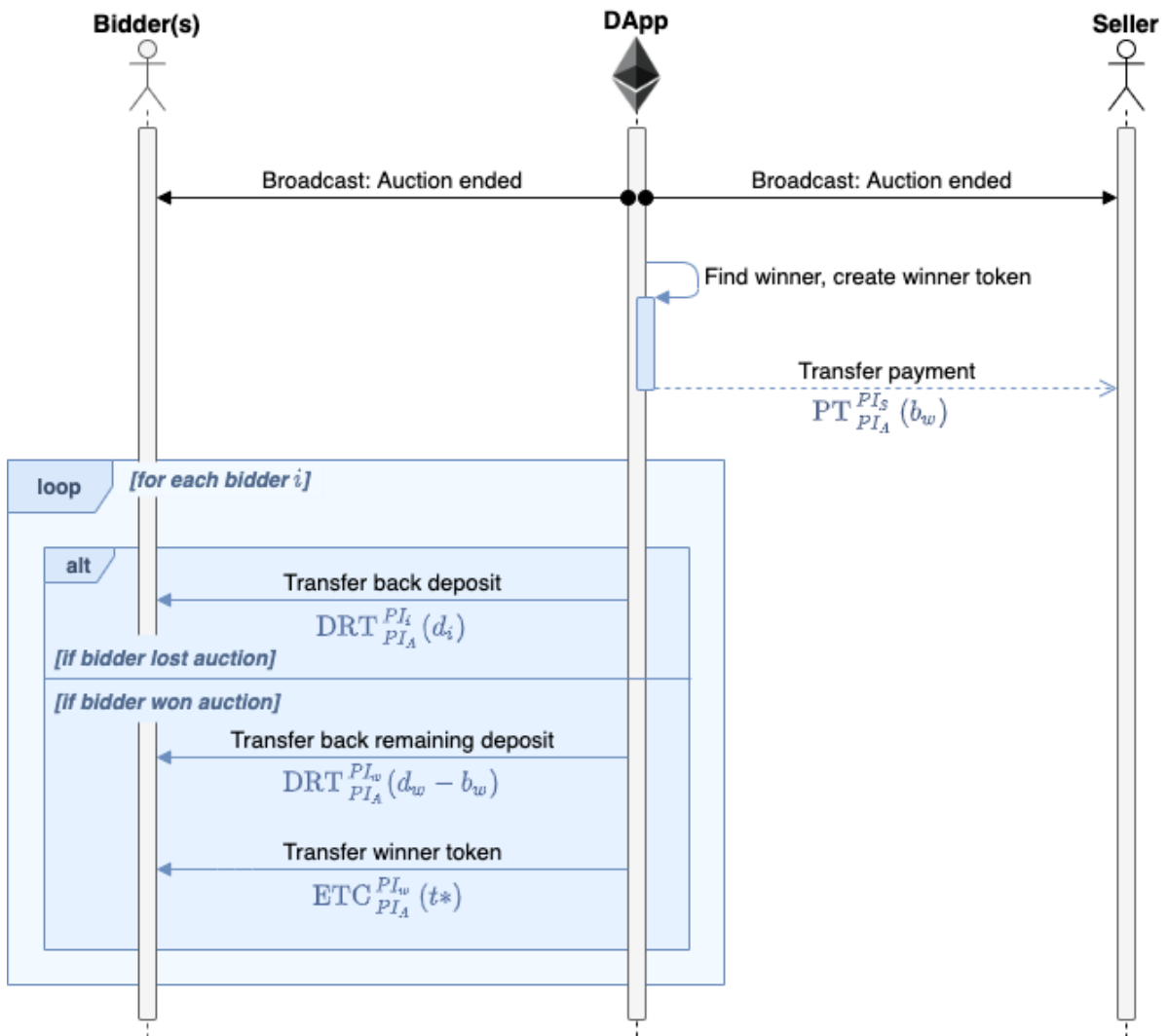


Figure 3.6: The DApp finds the auction winner, and completes the closing logic of the auction

and will be signed by the auction's secret key (Sk_A). It will also contain a boolean value that says if it has been spent or not to avoid double-spending. This value can only be changed once: when it is set to *true*, it cannot be changed back to *false*.

When the winner has been found, and the winner token has been created, the DApp will transfer the highest bid to the seller (Payment Transaction). Then it will loop through each bidder who has a stored open bid (those who did not forfeit their deposit) and transfer back their deposit with a Deposit Return Transaction. The bidder who won the auction will receive his remaining deposit after the bid has been subtracted from it ($d_w - b_w$). The winner will also receive the winner token with an Energy Transfer Commission.

3.2.5 Phase 5: Delete Auction

Finally, when the auction is finished, it will be deleted by the DApp. This, and all the other phases, will be more carefully explained in Chapter 4: Implementation.

Chapter 4

Implementation

This chapter will provide detailed information about the implementation of the solution. We will start the chapter by introducing the platform architecture, the actors, and the system components: the DApp User Interface, the Auction Controller, and the Auction entity. The latter two are smart contracts located in the Ethereum network. We will then describe the state variables of both contracts. As storing state variables is expensive in terms of gas, we will justify why they are included, why they are of a certain type and visibility, and explain their functionality. Finally, we will present the code of the two contracts step by step through the auction phases introduced in Chapter 3. The complete source code of the Auction Controller and Auction can be found in Appendix A and Appendix B respectively, and is also published to GitHub: <https://github.com/cristinatorp/master-thesis>

4.1 Platform Architecture

In the previous chapter, the auction system was introduced simply as a Decentralized Application. In reality, the platform is divided into three core components: the DApp UI, an Auction controller, and the Auction entity. The architecture of our system implementation is demonstrated in Figure 4.1.

The DApp UI is what the users of the system (namely, the sellers and the bidders) interact with. The Auction Controller and Auction entity are two separate smart contracts located in the Ethereum blockchain. The UI can communicate with these contracts

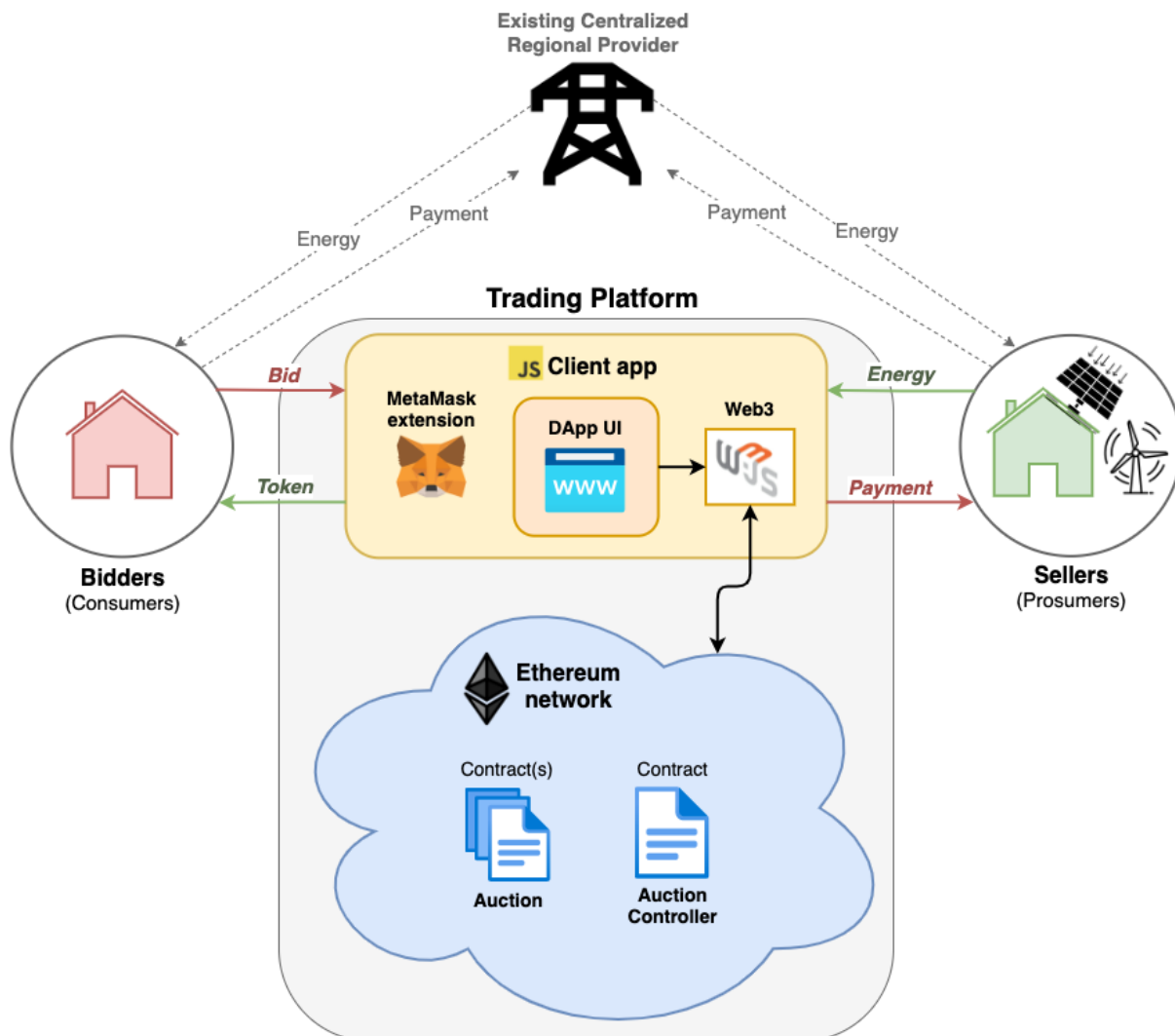


Figure 4.1: System architecture

through web3.js, which is "a collection of libraries which allow you to interact with a local or remote *Ethereum node*" [29]. The users will be able to transfer money to the contracts by using MetaMask [30], a cryptocurrency wallet that is available as a browser extension and that is integrated within the DApp UI.

The Auction Controller contract lives on the Ethereum blockchain and is responsible for *deploying* instances of the Auction contract to the network, and for deleting them when the individual auctions are finished. The Auction contract is responsible for controlling one specific auction. Several of these contracts can be deployed at any one point in time, each controlling its own auction entity. These three components together form the Decentralized Application for the Smart Energy Trading Platform.

4.1.1 Actors and Platform Components

In order to explain how the users interact with the DApp in detail, we have created several sequence diagrams where the two user types (*bidder* and *seller*) are denoted as *actors*, the DApp UI as a *boundary*, the Auction Controller as a *controller* and the Auction as an *entity*. These are represented at the top of each diagram and are displayed in Figure 4.2.



Figure 4.2: UML-symbols of the actors and components included in the Smart Trading Platform implementation

Seller: The seller represents the user (actor) that opens a new auction. He owns an excess amount of renewable energy that needs to be auctioned off.

Bidder: The bidder represents a user (actor) who bids on an auction. All interactions which stems from this user can be repeated from several different bidding accounts, as shown in Figure 3.4 and Figure 3.5.

DApp UI: The DApp UI is a system boundary that represents the frontend website with which the actors interact. It is included in the platform to make the platform user-friendly and abstracts away the complicated blockchain technology behind the application. The UI can listen to events emitted from the contracts and use the logged information in the events to display changes to the users.

Auction Controller: The Auction Controller is the auction manager. A controlling entity "*organizes and schedules the interactions between the boundaries and entities and serves as the mediator between them*" [31]. It is a smart contract that deploys new auction contracts, keeps a list of all active auctions, and deletes completed auctions. Because the DApp UI can listen to events emitted from both the controller and each auction instance, it will only interact with the controller to create or delete an auction; after an auction has been deployed, the DApp UI can interact directly with the Auction Entity for the intermediary steps. It is not possible, however, to create auctions without the controller. Moreover, the controller cannot be deleted and will live on the Ethereum Network forever.

Auction: The Auction is an entity that represents system data. It handles all the data for one individual auction. These contracts will be deployed from the Auction Controller and will only live throughout the duration of the auction.

4.2 Code Implementation

The implementation comprises the Auction Controller contract and the Auction contract. The DApp UI is only conceptual, and its implementation is therefore out of scope. This section will first introduce all state variables in the aforementioned contracts, and then demonstrate how the solution has been translated to code using the Solidity language, advancing through the five phases.

4.2.1 Auction Controller State Variables

The Auction Controller does not need many variables in order to manage the auction instances. In fact, there are only two state variables in the code and two events that will be logged. As they are expensive to store, most of the information is only stored in the auction instance to avoid redundancy and reduce cost. The variables are presented in Code Fragment 4.1 and described below.

```
1 address private admin;
2 // auction address => seller address
3 mapping(address => address) public sellerAddresses;
4
5 event AddedNewAuction(address auction);
6 event DeletedAuction(address auction);
```

Code Fragment 4.1: Auction controller state variables

State variables

- **admin:** This is the Ethereum account *address* of the account that deploys the Auction Controller. It is set to *private*, because it is not relevant to the users of the system to see which account is the admin account. This variable is included in the contract so the admin can delete auction contracts if the seller forgets or simply does not care to do so; everyone should not have permission to delete auction instances.
- **sellerAddresses:** This variable is a *mapping* and takes a key and a value. The key is the Ethereum account address of an auction (address of a contract), while

the value is the Ethereum account address of that auction's seller. The mapping is *public*, which means that everyone can see the ongoing auctions. It is included so the controller can keep track of all deployed auctions.

Events

- `AddedNewAuction`: Logs the Ethereum *address* of the new auction contract.
- `DeletedAuction`: Logs the Ethereum *address* of the deleted auction contract.

4.2.2 Auction State Variables

Contrary to the auction controller, the auction instance has many helpers and variables to control all aspects of an auction. The helpers are listed in Code Fragment 4.2 and the state variables are listed in Code Fragment 4.3. Most of the information about the auction is bundled into structs. Because structs are packed tightly [32], this layout will consequently save a significant amount of gas compared to storing each field by itself.

```
1 enum State {
2     ReadyForHiddenBids ,
3     ReadyForOpenBids ,
4     Closed ,
5     ReadyForDeletion
6 }
7
8 modifier inState(State expectedState) {
9     require(auctionInfo.currentState == expectedState, "Invalid ...
10         state");
11     -;
12 }
13
14 modifier isBeforeDeadline(uint deadline) {
15     require(block.timestamp < deadline, "Cannot bid after ...
16         deadline");
17     -;
18 }
19
20 modifier isAfterDeadline(uint deadline) {
21     require(block.timestamp > deadline, "Cannot perform this ...
22         action before the deadline");
23     -;
24 }
```



```
22
23 event CreatedNewAuction(AuctionInfo auctionInfo, uint ...
    currentTime);
24 event ReceivedHiddenBid(address bidder, uint deposit, uint ...
    currentTime);
25 event ReceivedOpenBid(address bidder, uint bid, uint currentTime);
26 event ClosedRound(string whichRound, State state, uint ...
    currentTime);
27 event ClosedAuctionWithNoBids(string whichRound);
28 event FoundHighestBid(Winner winner, uint currentTime);
29 event AuctionEnded(Winner winner, uint contractBalance, uint ...
    currentTime);
30 event TransferEvent(string context, address to, uint value, ...
    uint currentTime);
31 event RetrievedToken(address retrievedBy, uint currentTime);
```

Code Fragment 4.2: Auction helpers

Enums

- **State:** Enums in Solidity are explicitly convertible to an integer. As this enum has four members, the integers are in the range of 0–3. State’s members are the four different states the auction can be in: (1) Ready For Hidden Bids; (2) Ready For Open Bids; (3) Closed; and (4) Ready For Deletion. It will default to its first member when initiated.

Modifiers

- **inState:** This modifier will receive a State argument that represents the expected state, and will check that the current state is the expected state. If the states are unequal, the modifier will revert the function call.
- **isBeforeDeadline:** This modifier verifies that the current timestamp is before the deadline specified in the argument. It is used on the bid functions during the bidding process.
- **isAfterDeadline:** This is the opposite; it verifies that the current blockstamp is after the given deadline. This modifier is used when closing bidding rounds and when retrieving the token. These two timing modifiers are included to restrict when different functions can be called to ensure that they are only called at legitimate moments.

Events

- **CreatedNewAuction:** Emitted in the Auction's constructor, logs all the information in the *AuctionInfo* struct and the timestamp of creation.
- **ReceivedHiddenBid:** Emitted each time a hidden bid is received, logs the account address of the bidder, the deposit value, and the timestamp of the bid.
- **ReceivedOpenBid:** Emitted each time an open bid is received, logs the account address of the bidder, the open bid value, and the timestamp of the bid.
- **ClosedRound:** Emitted when any of the two bidding rounds close. Logs a string describing either "Hidden round" or "Open round", the current state of the auction, and the current timestamp.
- **ClosedAuctionWithNoBids:** Emitted if the auction closes without bids (hidden round: no bids received, open round: no valid bids received). Logs which round the auction closed and the current timestamp.
- **FoundHighestBid:** Emitted when the auction winner is declared. Logs the *Winner* struct and the current timestamp.
- **AuctionEnded:** Emitted when the auction is completely finished. Logs the *Winner* struct again, the contract's balance, and the current timestamp.
- **TransferEvent:** Emitted for each transfer event: transferring back deposits, transferring the highest bid to the seller, and transferring remaining deposits to the seller. Logs the context of the transfer, which address is receiving Ether, the value that is transferred, and the current timestamp.
- **RetrievedToken:** Emitted when the winner retrieves their token. Logs the address that retrieved it and the current timestamp.

```
1 struct AuctionInfo {
2     State currentState;
3     address payable seller;
4     uint energyAmount;
5     uint minBidValue;
6     uint depositValue;
7     uint hiddenBidsDeadline;
8     uint openBidsDeadline;
9 }
10
11 struct Bid {
12     bool existsHiddenBid;
13     bytes32 hiddenBid;
```

```
14     uint openBid;
15     bool isOpenBidValid;
16     uint deposit;
17 }
18
19 struct Winner {
20     address accountAddress;
21     uint bid;
22 }
23
24 struct Token {
25     address winner;
26     address auctionContract;
27     uint energyAmount;
28     uint validUntil;
29     bool isSpent;
30 }
31
32 address private controller;
33 AuctionInfo public auctionInfo;
34 Winner public winner;
35 mapping(address => Bid) public bids;
36 mapping(address => Token) private token;
37 address[] public hiddenBidsAddresses;
```

Code Fragment 4.3: Auction state variables

Structs

- **AuctionInfo:** This struct bundles up all the information about the auction. Every field except for its state is static and will be set in the Auction's constructor.
 - **currentState:** Is of type *State* (an *enum*) and will be updated throughout the Auction's lifecycle. Necessary for verifying function access and for emitting events which the DApp UI can listen for.
 - **seller:** Stores the seller's account *address*. This is a *payable* address because it must be possible to transfer ether to the seller.
 - **energyAmount:** The amount of energy that is auctioned off by the seller. The field is a plain *uint*. Must be stored for the token information.
 - **minBidValue:** The minimum bid value that the seller will accept for this amount of energy. Will be stored as *wei*, which is implicitly a *uint*.

- **depositValue**: The minimum deposit value which every bidder must transfer to the contract in order to participate in the auction. Will also be stored as *wei*.
- **hiddenBidsDeadline**: A `uint` that defines when the hidden bid round closes. Stated as seconds.
- **openBidsDeadline**: A `uint` that defines when the open bid round closes. Stated as seconds.
- **Bid**: Bundles up all information about a single bid.
 - **existsHiddenBid**: A `bool` that states if a hidden bid exists. This field is necessary because the `Bid` struct will be given as a value to a *mapping* and will always return a value, even if the key does not exist. If called on an invalid key, the struct will contain every field with default values. Therefore, this value will return as false for all invalid keys; for valid keys, however, this will return true.
 - **hiddenBid**: Will contain a hashed bid, and is therefore of type *bytes32*.
 - **openBid**: Contains the open, revealed bid in *uint* (as *wei*).
 - **isOpenBidValid**: A `bool` that defines if the open bid is valid or not. Necessary for verifying whether the bidder should receive their deposit back or not.
 - **deposit**: Contains the deposit amount of a bidder given as a *uint* (in *wei*).
- **Winner**: Bundles up information about the winner of the auction.
 - **accountAddress**: Stores the account *address* of the auction winner.
 - **bid**: Stores the highest open bid received throughout the auction.
- **Token**: The token that can be retrieved by the winner at the end of the auction. This can be used as the verification for the winner when obtaining his/her energy (how the winner retrieves energy is out of scope).
 - **winner**: Stores the auction winner's account *address*.
 - **auctionContract**: Stores the auction contract's *address*.
 - **energyAmount**: This contains the amount of energy that the winner won in the auction as a *uint*.
 - **validUntil**: The timestamp when the token is no longer valid, given in seconds since Unix epoch (*uint*).
 - **isSpent**: A `bool` value that declares if the token has been spent or not. This is included to avoid double spending. When the value is changed to *true*, it cannot be changed back to *false*.

State variables

- **controller**: The contract *address* of the auction controller. Set to *private*, as it is only used for restricting who should be able to delete the auction instance.
- **auctionInfo**: The auction information bundled up in the struct *AuctionInfo*.
- **winner**: The information about the winner, of struct *Winner*.
- **bids**: A *mapping* of all bids received throughout the bidding process. Key: the bidder's account address, value: *Bid* struct.
- **token**: A *mapping* of the token information. Key: the winner's account address, value: *Token* struct. This mapping is set to *private*; no one except for the winner should be able to get this information. A getter is given for retrieving the token with appropriate verification (see subsection 4.2.6).
- **hiddenBidsAddresses**: An *array* of all account addresses that have bid in the hidden round. This is necessary to loop through the bids in later functions, as it is not possible to find the length of a mapping in Solidity.

4.2.3 Phase 1: Create Auction

Figure 4.3 provides an overview of the interactions that take place in the system when a new auction is created. The seller initiates the process by using the DApp UI to create an auction. He will enter the amount of energy to be auctioned off, the minimum bid value, and the deposit value each bidder must transfer in order to participate in the auction (both values will be given as ETH). The deposit value should be significantly higher than what the seller expects to receive as the highest bid because the winner's bid will be subtracted from his deposit amount at the end of the auction. If the highest bid turns

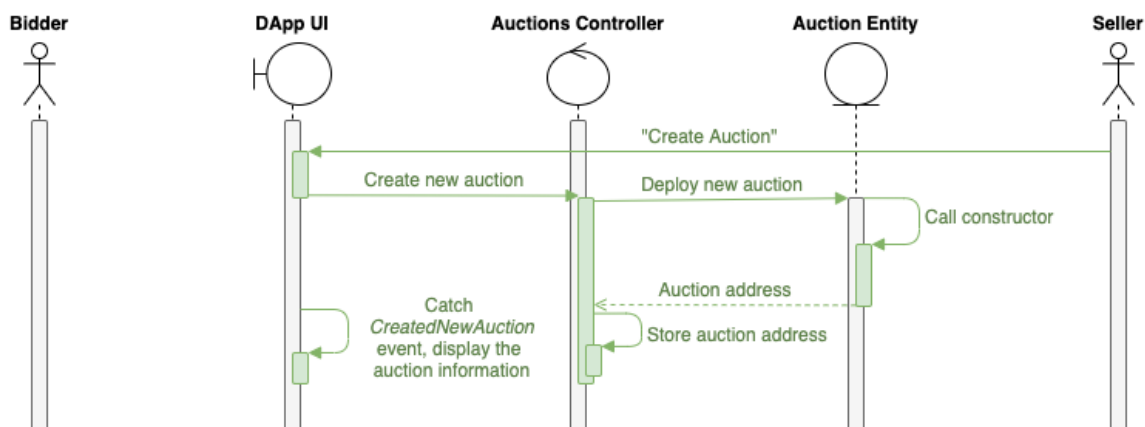


Figure 4.3: Phase 1: No state (creation)

out to be higher than the deposit amount in the reveal phase, the seller will still only receive the winner's deposit, as there will not be any excess ETH on the contract. The UI will show the average value of earlier auctions' winning bids for similar amounts of energy to make this decision easier for the seller.

When the UI contacts the Auction Controller to create a new auction, it is the controller's `deployNewAuction()` function that is called. The UI relays the seller's account address and the energy amount, the minimum bid value, and the minimum deposit value that the seller entered. This function is presented in Code Fragment 4.4. By using `new Auction()`, the constructor of the Auction smart contract will be called and thus be deployed to the Ethereum network. The controller will save the address returned from the call as a new entry in the mapping `sellerAddresses` with the auction address as its key and the seller address as its value. Finally, it will fire the `AddedNewAuction` event that will be visible in the transaction receipt on the Ethereum blockchain.

```
1 function deployNewAuction(  
2     address payable _seller,  
3     uint _energyAmount,  
4     uint _minBidValue,  
5     uint _depositValue  
6 ) public {  
7     // Deploy new auction contract  
8     Auction newAuction = new Auction(  
9         _seller,  
10        _energyAmount,  
11        _minBidValue,  
12        _depositValue  
13    );  
14  
15    // Save seller address  
16    sellerAddresses[address(newAuction)] = _seller;  
17    emit AddedNewAuction(address(newAuction));  
18 }
```

Code Fragment 4.4: Auction controller: deploy a new auction

The constructor of the Auction contract will save the initial information about the auction (see Code Fragment 4.5). It gets the current time from `block.timestamp` and saves it as a local variable (will be stored in memory until the function is complete, then

discarded) and stores the address of the controller in its state variable. The information about the auction is stored as an `AuctionInfo` struct. The state will be initialized to `ReadyForHiddenBids`, and the values received as arguments will be saved to their respective fields (notice that the two value fields are given the suffix `wei`). The two deadlines, `hiddenBidsDeadline` and `openBidsDeadline` will be set to 24 hours from now and 48 hours from now, respectively. `block.timestamp` returns the "current block timestamp as seconds since unix epoch" [28]. Solidity offers convenience properties for converting time units based in seconds to numbers. Adding `1 days` and `2 days` to `block.timestamp` will thus give us the deadlines in seconds. Finally, the auction will fire an event with the aforementioned information and the timestamp. The DApp UI will catch this event and use the information in the event log to update its list of active auctions and subsequently display it to its users.

```
1 constructor(  
2     address payable _seller,  
3     uint _energyAmount,  
4     uint _minBidValue,  
5     uint _depositValue  
6 ) {  
7     uint currentTime = block.timestamp;  
8     controller = msg.sender;  
9     auctionInfo = AuctionInfo({  
10        currentState: State.ReadyForHiddenBids,  
11        seller: _seller,  
12        energyAmount: _energyAmount,  
13        minBidValue: _minBidValue * 1 wei,  
14        depositValue: _depositValue * 1 wei,  
15        hiddenBidsDeadline: currentTime + 1 days,  
16        openBidsDeadline: currentTime + 2 days  
17    });  
18  
19    emit CreatedNewAuction(auctionInfo, currentTime);  
20 }
```

Code Fragment 4.5: The constructor of the auction smart contract

4.2.4 Phase 2: Bid in the Hidden Round

The auction entity will wait for hidden bids for 24 hours (see Figure 4.4). Potential bidders will use the DApp UI to find the auction they want to bid on, and will select bids appropriately. As an incentive to not bid lower than what the minimum bid value is stated as, there will be a clear warning for the bidders that if the bid is revealed to be invalid in the open round, the deposit will be transferred to the seller instead of returned back to the bidder upon closing the auction. The same is true if the hashed bid is not equal to the open bid in the reveal. Upon submitting the bid, the UI will hash the bid value along with a salt value chosen by the bidder (a mnemonic phrase, for instance) and transfer the hashed bid and the deposit directly to the auction contract. The bidders' Ethereum wallets will be connected to the UI by using the MetaMask browser extension, and the deposit will be withdrawn from the account connected to MetaMask.

The auction contract receives a hidden bid from the UI when the UI calls the `bidInHiddenRound()` function (presented in Code Fragment 4.6). The single argument `bid` is hashed from the frontend with the `soliditySha3` function of the **web3** library, which is the recommended library for creating JS applications that connect to the Ethereum blockchain. The function has two modifiers: `inState` and `isBeforeDeadline`.

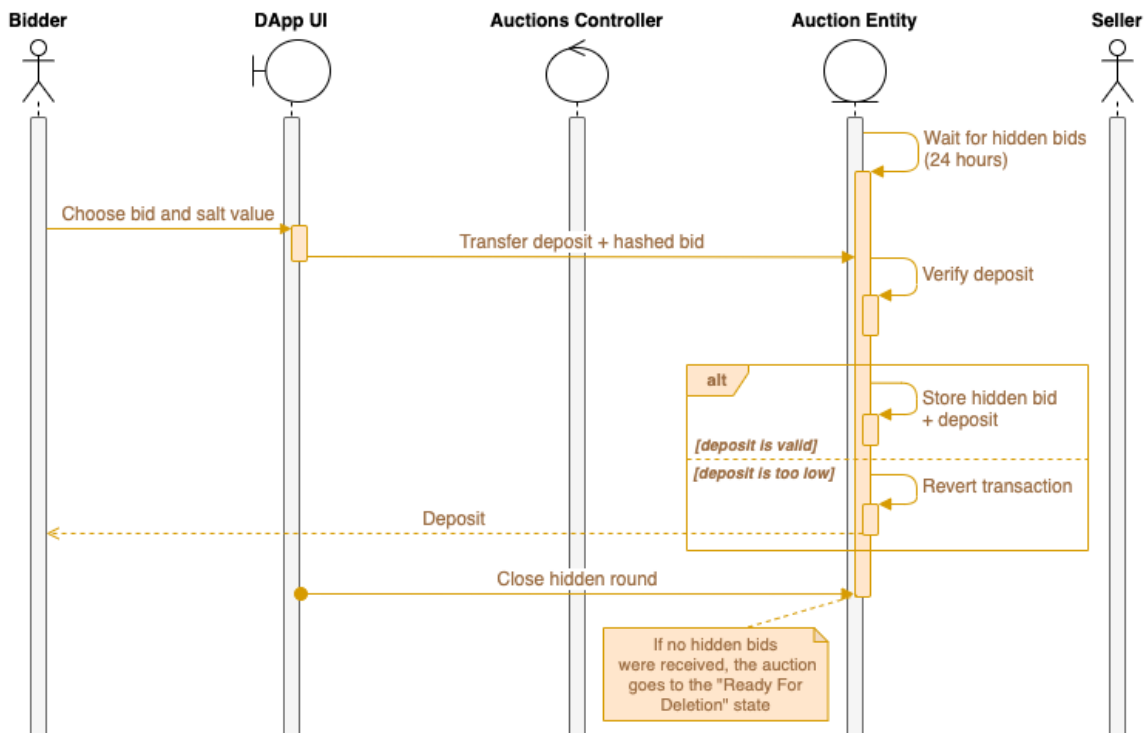


Figure 4.4: Model phase 2: Ready for hidden bids

It is required that the state is `ReadyForHiddenBids` and that the function is called before the deadline for the hidden round is over (approximately 24 hours from creation). As the function is public and can be called at any time, these are added as safeguards to ensure that the function will revert if a bidder tries to bid on the auction after it has proceeded to the next round. Additionally, another verification check is called at the beginning of the function that validates that the transferred deposit amount is equal to or more than the minimum deposit amount required. This `require` statement will revert the transaction if the condition yields false.

```
1 function bidInHiddenRound(bytes32 bid) public payable
2     inState(State.ReadyForHiddenBids)
3     isBeforeDeadline(auctionInfo.hiddenBidsDeadline)
4 {
5     require(msg.value >= auctionInfo.depositValue,
6         "Deposit value is too low");
7
8     bids[msg.sender] = Bid({
9         existsHiddenBid: true,
10        hiddenBid: bid,
11        openBid: 0,
12        isOpenBidValid: false,
13        deposit: msg.value * 1 wei
14    });
15
16    hiddenBidsAddresses.push(msg.sender);
17    emit ReceivedHiddenBid(
18        msg.sender,
19        msg.value,
20        block.timestamp
21    );
22 }
```

Code Fragment 4.6: Bid in hidden round

If all checks are passed, the bid will be stored as an entry in the mapping `bids` with the bidder's Ethereum account address as the key and a `Bid` struct as the value. When this struct is initialized, we set `existsHiddenBid` to `true`, `hiddenBid` to the hashed bid received from the UI, and `deposit` to the message value received converted to `wei`. The last two fields are set to their default values as they are not yet relevant. The bidder's

account address will also be saved in the array `hiddenBidsAddresses`. Finally, the `ReceivedHiddenBid` event will be fired, which logs the account address of the bidder, the deposit value, and the current timestamp.

Considering that keys in mappings cannot be repeated, one single bidder cannot bid several times; however, if the bidder wishes to bid again, he may use another account in his Ethereum wallet to do so. When the 24 hours have elapsed, the DApp UI will call the auction's `closeHiddenRound()` function (Code Fragment 4.7).

```
1 function closeHiddenRound() public
2     inState(State.ReadyForHiddenBids)
3     isAfterDeadline(auctionInfo.hiddenBidsDeadline)
4 {
5     if (hiddenBidsAddresses.length == 0) {
6         auctionInfo.currentState = State.ReadyForDeletion;
7         emit ClosedAuctionWithNoBids("Hidden round", ...
8             block.timestamp);
9     } else {
10        auctionInfo.currentState = State.ReadyForOpenBids;
11        emit ClosedRound(
12            "Hidden round",
13            auctionInfo.currentState,
14            block.timestamp
15        );
16    }
```

Code Fragment 4.7: Close hidden round

Closing the hidden round requires that the auction is still in the state `ReadyForHiddenBids`. It also requires that the function is called *after* the hidden bids deadline. If these two tests are passed, the function checks how many bids were received in the hidden round by checking the length of the array `hiddenBidsAddresses`. The auction closes if none were received; the state will be changed to `ReadyForDeletion` and the `ClosedAuctionWithNoBids` event will fire. It will then proceed to phase 5. If the length is above zero, however, the auction state will be changed to `ReadyForOpenBids` and will proceed to the next phase. The event `ClosedRound` will fire and log the current round, state, and timestamp. Some could assume that if there were only one bidder, this bidder would automatically become the winner; however, because the bid is hashed,

there is no way of knowing if the bid is actually valid or not. The auction must therefore proceed to the next round, and thus the bidder must re-enter his bid in plaintext in the open round in order to reveal it.

4.2.5 Phase 3: Bid in the Open Round

The auction will wait for open bids for 24 hours (see Figure 4.5). When the open round begins, the UI will push a notification to all bidders who participated in this particular auction's hidden round to remind them to also bid in the open round. They will lose their deposit if they forget to do so. As the UI keeps track of which auctions the signed-in user has already bid on, the auction will be easy to find in the bidder's auction list. The bidder can now choose this auction and re-enter their bid to reveal it. To incentivize the bidders to not alter their bid, it will again be stated clearly in the UI that the bidders will lose their deposit when the contract detects an invalid bid or a fraudulent attempt.

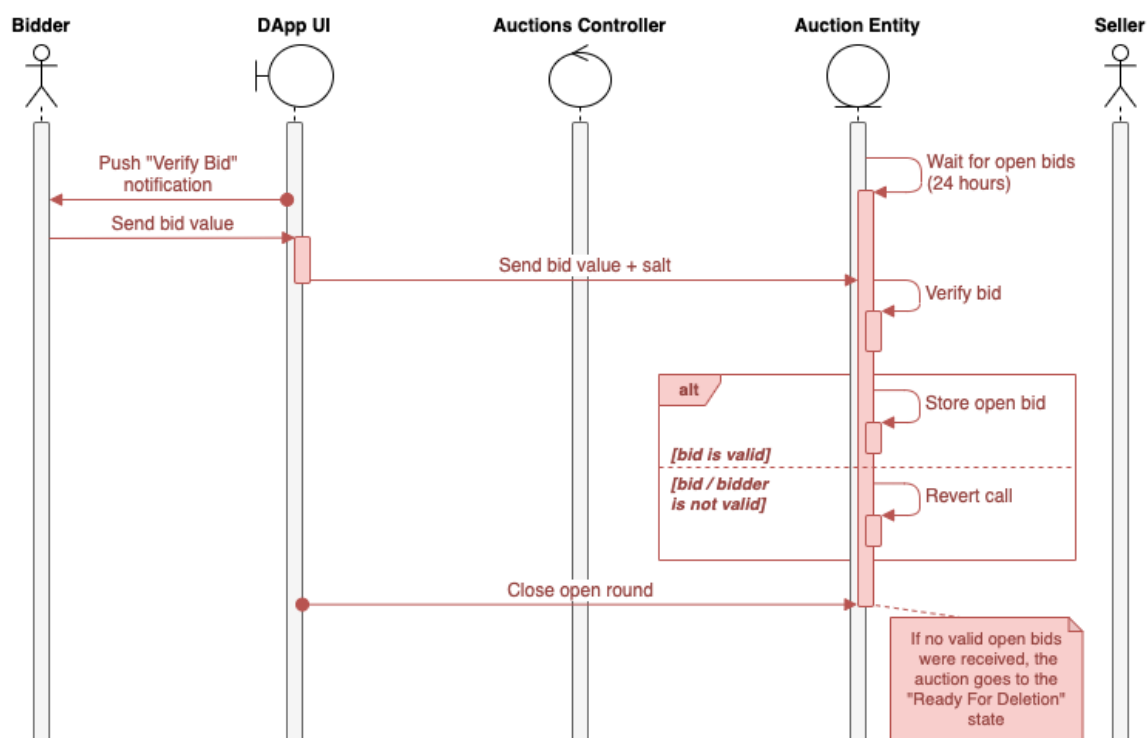


Figure 4.5: Model phase 3: Ready for open bids

When bidders reveal their bid value by sending their open bid through the UI, the UI calls the auction's `bidInHiddenRound()` function (Code Fragment 4.8). It is only allowed to call this function when the auction is in the `ReadyForOpenBids` state and before the open bids deadline. The UI passes the open bid and the salt which this bid was hashed with during the hidden bid round as arguments to the function.

Several `require` statements are then called inside the function body: it verifies that the boolean value `existsHiddenBid` is `true` in the struct returned from the mapping `bids` with this bidder's account address and that the open bid received is higher than the minimum bid value set as a requirement by the seller. The last check is to validate that the bidder's open bid is equal to his hidden bid. This is done by hashing the open bid with the given salt and comparing this hash value with the hidden bid (which is also a hash value). The `keccak256(abi.encodePacked(<bid>, <salt>))` will return the same hash as `web3.utils.soliditySha3(<bid>, <salt>)` that is used in the JS code, as they are aliases of the same hashing function. If either of these three `require` statements fails (in addition to the two modifiers), the function will revert, and the rest of the function will be skipped.

```

1 function bidInOpenRound(uint openBid, string memory salt) public
2   inState(State.ReadyForOpenBids)
3   isBeforeDeadline(auctionInfo.openBidsDeadline)
4 {
5   require(bids[msg.sender].existsHiddenBid,
6     "This account has not bidden in the hidden round");
7   require(openBid >= auctionInfo.minBidValue, "Bid value is ...
8     too low");
9
10  bytes32 hashedBid = keccak256(abi.encodePacked(openBid, ...
11    salt));
12
13  require(bids[msg.sender].hiddenBid == hashedBid,
14    "Open bid and hidden bid do not match");
15
16  bids[msg.sender].isOpenBidValid = true;
17  bids[msg.sender].openBid = openBid;
18  emit ReceivedOpenBid(msg.sender, openBid, block.timestamp);
19 }

```

Code Fragment 4.8: Bid in open round; revealing bids

After ensuring that the bid is valid, the rest of the Bid struct will be updated. `isOpenBidValid` is set to true and `openBid` receives the `openBid` as a `uint` value. This implicitly means that if the bid is invalid, `isOpenBidValid` will still yield false. Finally, as in the other functions, an appropriate event will be fired: `ReceivedOpenBid`, which logs the bidder address, the bid, and the current timestamp.

```
1 function closeAuction() public
2     isAfterDeadline(auctionInfo.openBidsDeadline)
3     isInState(State.ReadyForOpenBids)
4 {
5     uint validOpenBids = 0;
6     for (uint i = 0; i < hiddenBidsAddresses.length; i++) {
7         if (bids[hiddenBidsAddresses[i]].isOpenBidValid) {
8             validOpenBids += 1;
9         }
10    }
11
12    if (validOpenBids == 0) {
13        auctionInfo.currentState = State.ReadyForDeletion;
14        emit ClosedAuctionWithNoBids(
15            "Open round, no valid bids",
16            block.timestamp
17        );
18    } else {
19        auctionInfo.currentState = State.Closed;
20        emit ClosedRound(
21            "Open round",
22            auctionInfo.currentState,
23            block.timestamp
24        );
25
26        findWinner();
27    }
28 }
```

Code Fragment 4.9: Close auction

After the 24 hours of the open round has expired the DApp UI will call the auction contract's `closeAuction()` function (Code Fragment 4.9). This function cannot be called unless the open bids deadline has expired or if the auction's state is not

ReadyForOpenBids. If the auction did not receive any valid bids during the open round, there is no winner; the auction will proceed to its ReadyForDeletion state and will fire a ClosedAuctionWithNoBids event, which the UI will catch and subsequently proceed to delete the auction contract. To calculate the number of valid open bids, we will loop through each bid in the bids mapping by using the index of the array hiddenBidsAddresses. If any of the bids has a true value of isOpenBidValid, the local uint variable validOpenBids will increment by one. If it is equal to 0 when the loop finishes, there are no valid bids. If it is above 0, valid bids exist, and a winner must be found. The auction state is updated to Closed, and the ClosedAuction event is fired, which logs the current round, the current state of the auction, and the current timestamp. It then calls findWinner() to proceed with the closing logic.

4.2.6 Phase 4: Close Auction

When the auction closes, the auction entity will loop through the stored open bids to find the highest value (see Figure 4.6). The findWinner() function presented in Code

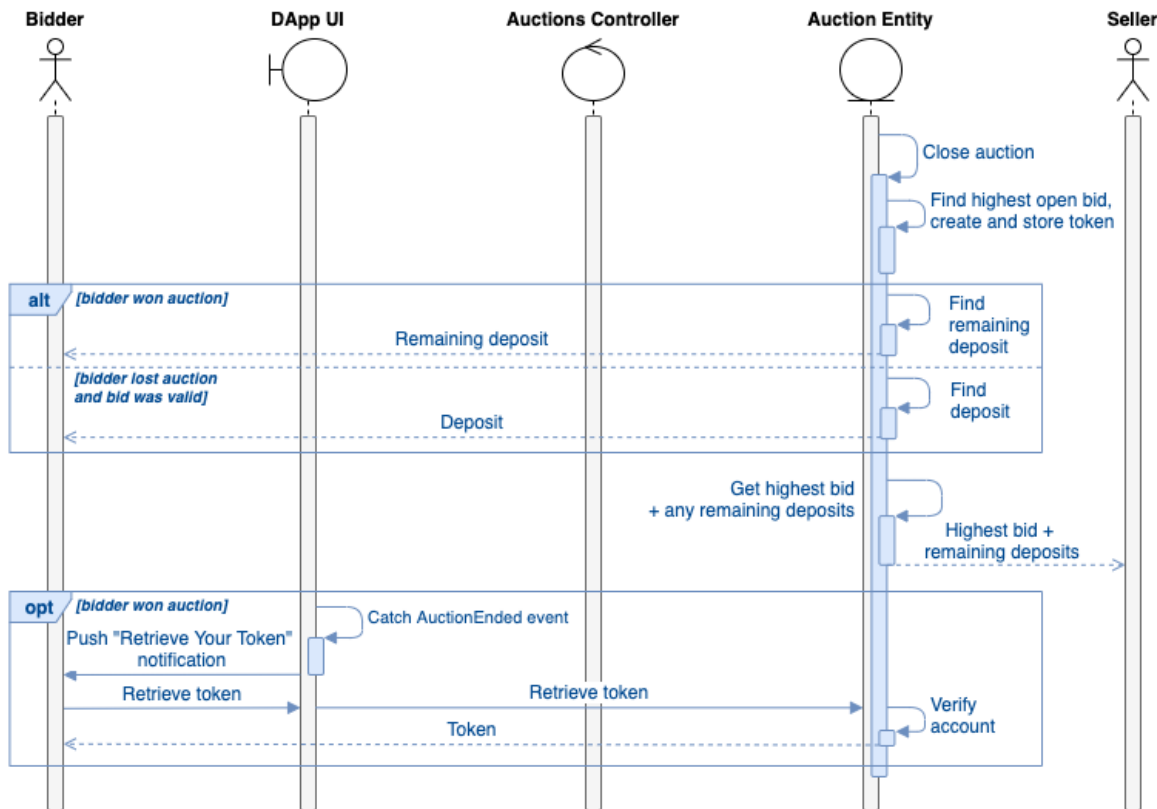


Figure 4.6: Model phase 4: Close auction

Fragment 4.10 is *internal*, which means that it can only be called by the auction contract. It still has a modifier however, simply to rigidly uphold the contract's security and integrity; the function can only be called if the auction is in the `Closed` state. In the function body, two local variables are declared: `winnerAddress` (defaults to `0x00`) and `highestBid` (defaults to `0`). These are updated while we loop through all the bids.

```
1 function findWinner() internal inState(State.Closed) {
2     address winnerAddress;
3     uint highestBid;
4
5     for(uint i = 0; i < hiddenBidsAddresses.length; i++) {
6         address bidder = hiddenBidsAddresses[i];
7         if (!bids[bidder].isOpenBidValid) continue;
8         uint bid = bids[bidder].openBid;
9
10        if (bid > highestBid) {
11            winnerAddress = bidder;
12            highestBid = bid;
13        }
14    }
15
16    winner = Winner({
17        accountAddress: winnerAddress,
18        bid: highestBid
19    });
20    emit FoundHighestBid(winner, block.timestamp);
21
22    token[winnerAddress] = Token({
23        winner: winnerAddress,
24        auctionContract: address(this),
25        energyAmount: auctionInfo.energyAmount,
26        validUntil: block.timestamp + 12 weeks,
27        isSpent: false
28    });
29
30    transferBackDeposits();
31 }
```

Code Fragment 4.10: Find the auction winner

We loop through the bids by using the index of the array `hiddenBidsAddresses` and extract each open bid from the `Bid` struct. If the bid is above the current highest bid, we update the variables with the current bid's account address and open bid. When the loop is complete, we have consequently found the highest bid and its correlated account address. The state variable `winner` is then stored as a `Winner` struct with the winner's address and bid, and the `FoundHighestBid` event is fired which logs the `winner` struct and the current timestamp.

As Solidity does not support sending anything other than ETH to an account address, we must create and store the token in the contract state so the winner can retrieve it. We, therefore, create an entry in the `token` mapping: the key is set to the winner's account address, and the value is set to a new instance of the `Token` struct. The struct receives the winner's address, the address of the contract (for traceability), the amount of energy that the bidder won, and the timestamp when the token becomes invalid (approximately 12 weeks from its creation). Additionally, we will set the bool value `isSpent` to false. The function will then call the function `transferBackDeposits()` (presented in Code Fragment 4.11).

The `transferBackDeposits()` function is also *internal* and has the modifier `inState(State.Closed)`. Inside its function body it has another requirement: the account address of the `winner` struct must be initialized before continuing (checked by verifying that the address is not its default value).

If these tests are passed, the function loops through all the bids in the same manner as in earlier functions. If the bid's `isOpenBidValid` field is set to *false*, we skip to the next bid; this bidder will not regain his deposit. However, if the bid was valid, we check if the bidder is the auction winner; the winner will only receive the remaining value after subtracting his bid from his deposit. If the bid was higher or equal to his deposit, the remaining deposit is 0, and we skip to the next bidder. All bidders that lost the auction will receive their full deposit back. The `TransferEvent` will fire for each transfer and log the current context: "Transfer back deposit to bidder", the bidder's account address, the deposit value, and the current timestamp. When the loop is completed and all deposits transferred, we will proceed to the `transferHighestBidToSeller()` function.


```
1 function transferBackDeposits() internal inState(State.Closed) {
2     require(winner.accountAddress != address(0), "Must find a ...
3         winner before sending back deposits");
4
5     for (uint i = 0; i < hiddenBidsAddresses.length; i++) {
6         payable bidderAddress = ...
7             payable(hiddenBidsAddresses[i]);
8         Bid memory bid = bids[bidderAddress];
9
10        // Do not send back deposit to invalid bidders
11        if (!bid.isOpenBidValid) continue;
12
13        bool isWinner = bidderAddress == winner.accountAddress;
14        if (isWinner && bid.openBid >= bid.deposit) continue;
15        uint deposit = isWinner ? bid.deposit - bid.openBid : ...
16            bid.deposit;
17
18        emit TransferEvent(
19            "Transfer back deposit to bidder",
20            bidderAddress,
21            deposit,
22            block.timestamp
23        );
24
25        bidderAddress.transfer(deposit);
26    }
27
28    transferHighestBidToSeller();
29 }
```

Code Fragment 4.11: Transfer back deposits

The function `transferHighestBidToSeller()` (Code Fragment 4.12) will transfer the highest bid to the seller, but also the remaining deposits (if any). It extracts the highest bid and the seller address from the state variables. If the highest bid is higher than the deposit value given by the seller, the seller will only receive the deposit (remember that this information is given to the seller as a UI warning upon auction creation). A `TransferEvent` is fired and logs the correct context (if the seller receives the highest bid, it will be "Transfer highest bid to seller", if not it will be "The highest bid was higher

than the deposit value. Transferring the deposit to seller instead"), the seller account address as the to field, the transfer value, and the current timestamp.

```
1 function transferHighestBidToSeller() internal ...
  inState(State.Closed) {
2     uint highestBid = winner.bid;
3     address payable seller = auctionInfo.seller;
4     string memory eventMsg = "Transfer highest bid to seller";
5
6     if (highestBid > auctionInfo.depositValue) {
7         highestBid = auctionInfo.depositValue;
8         eventMsg = "The highest bid was higher than the deposit ...
          value. Transferring the deposit to seller instead";
9     }
10
11     emit TransferEvent(
12         eventMsg,
13         seller,
14         highestBid,
15         block.timestamp
16     );
17
18     seller.transfer(highestBid);
19
20     // Transfer deposits of invalid bidders to seller
21     uint contractBalance = address(this).balance;
22     if (contractBalance > 0) {
23         emit TransferEvent(
24             "Transfer contract balance to seller",
25             seller,
26             contractBalance,
27             block.timestamp
28         );
29
30         seller.transfer(contractBalance);
31     }
32
33     emit AuctionEnded(winner, address(this).balance, ...
34         block.timestamp);
}
```

Code Fragment 4.12: Transfer highest bid and remaining deposits to seller

After transferring the highest bid, we check the remaining contract balance. If this balance is higher than 0, there are still some deposits on the contract that did not get returned to bidders who sent an invalid bid during the bidding process. These deposits are transferred to the seller by simply transferring the remaining contract balance. Another `TransferEvent` is fired which logs the "Transfer contract balance to seller" context, the seller address, the value `contractBalance`, and the current timestamp. This action concludes all auction closing logic, and the event `AuctionEnded` fires and logs the `winner` struct, the remaining contract balance (should be 0), and the current timestamp. This event will be caught by the UI and used to push a notification to the winner's account that a token has been created for the winner and is ready to be retrieved.

When the winner account retrieves its token through the UI, the UI calls the auction's `retrieveToken()` function (presented in Code Fragment 4.13). Considering that this function is *public* (because the `token` state variable is private) there are rigid security checks that protects it. Two modifiers are present: `inState` that verifies that the auction is closed, and `isAfterDeadline` that verifies that the current timestamp is after the open bids deadline. Another require condition is included, which is the most important: the caller of the function must be the winner's account address. Without this check, everyone on the Ethereum network could access the winner's token.

```
1 function retrieveToken() public
2     inState(State.Closed)
3     isAfterDeadline(auctionInfo.openBidsDeadline)
4     returns(Token memory)
5 {
6     require(msg.sender == winner.accountAddress,
7         "You are not the winner of the auction!");
8
9     auctionInfo.currentState = State.ReadyForDeletion;
10    emit RetrievedToken(msg.sender, block.timestamp);
11
12    return token[msg.sender];
13 }
```

Code Fragment 4.13: Retrieve token

The state of the auction is set to `ReadyForDeletion` at this point as there is no logic left to be conducted. The `RetrievedToken` event is also fired, which logs the winner's

account address and the current timestamp. Finally, the function returns the `Token` struct to the winner.

4.2.7 Phase 5: Delete Auction

An auction can be deleted upon four occasions:

1. Auction has closed with no hidden bids
2. Auction has closed with no valid open bids
3. Token has been retrieved
4. Token has expired

The first three occasions are caught by the UI by listening for the events `ClosedAuctionWithNoBids` and `RetrievedToken`. The fourth occasion is exposed when the UI performs a daily cleanup where it loops through all the auctions and checks the timestamp of the `validUntil` field of the token stored in the auction's state variables and checks if the token has expired yet. This is illustrated in Figure 4.7.

If any of these four cases occur, then the UI will call the Auction Controller's `deleteAuction()` function in Code Fragment 4.14. The function takes the auction address as its single argument. It retrieves the auction instance that resides at this address and then verifies the caller of the function: only the admin or the seller of the auction is allowed to delete the auction instance. It checks if the token has expired by looking up the `validUntil` field of the token: if it is 0 it has not been set yet, and the auction is still ongoing (token has not expired). If the current timestamp is more than the `validUntil` field, the token has expired. If it has expired, the controller proceeds to delete the auction. If it has not expired yet, the controller checks the other conditions by checking the

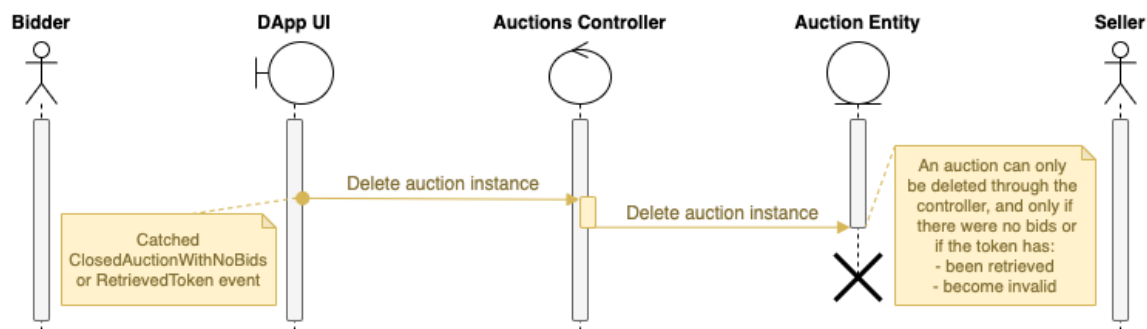


Figure 4.7: Model phase 5: Ready for deletion

state of the auction. If it is `ReadyForDeletion`, the auction has closed with no (valid) bids, and the deletion can proceed.

The controller calls the auction's `deleteAuction()` function (Code Fragment 4.15) and deletes the entry from its mapping `sellerAddresses` by its key (the auction address). Finally, it fires the event `DeletedAuction` which logs the address of the auction contract that was deleted.

```

1  /// Auction can only be deleted by admin or by the auction seller
2  function deleteAuction(address auctionAddress) public {
3      Auction auction = Auction(auctionAddress);
4
5      require(msg.sender == sellerAddresses[auctionAddress]
6             || msg.sender == admin,
7             "Can only be deleted by admin or the auction seller");
8
9      bool tokenExpired = block.timestamp > ...
10         auction.getTokenValidUntil()
11         && auction.getTokenValidUntil() != 0;
12     if (!tokenExpired) {
13         require(auction.getCurrentState() == ...
14                Auction.State.ReadyForDeletion,
15                "Cannot delete auction before the token has expired or ...
16                been retrieved");
17     }
18
19     auction.deleteAuction();
20     delete sellerAddresses[auctionAddress];
21
22     emit DeletedAuction(auctionAddress);
23 }

```

Code Fragment 4.14: Auction controller: delete auction

The auction instance's `deleteAuction` function is *external* which means that the auction contract cannot delete itself. It must be called from the controller contract. If someone else tries to delete the auction, the function will revert with the error message *"You are not allowed to delete this auction!"*. When the controller deletes the auction, it will **selfdestruct** and return the remaining contract balance (which should be 0, as we transferred this in `transferHighestBidToSeller()`) to the seller address. Note that

`selfdestruct` does not actually delete the contract from the blockchain: it resets all the state variables to their default values and makes all functions uncallable. All calls and transactions, as well as the contract state, will be available in the blockchain history and thus further ensures transparency. The contract will only be "deleted" from the blockchain *future*. Finally, the UI will move the auction from its list of active auctions to its archive.

```
1 function deleteAuction() external {
2     require(msg.sender == controller,
3         "You are not allowed to delete this auction!");
4     selfdestruct(auctionInfo.seller);
5 }
```

Code Fragment 4.15: Delete auction

Chapter 5

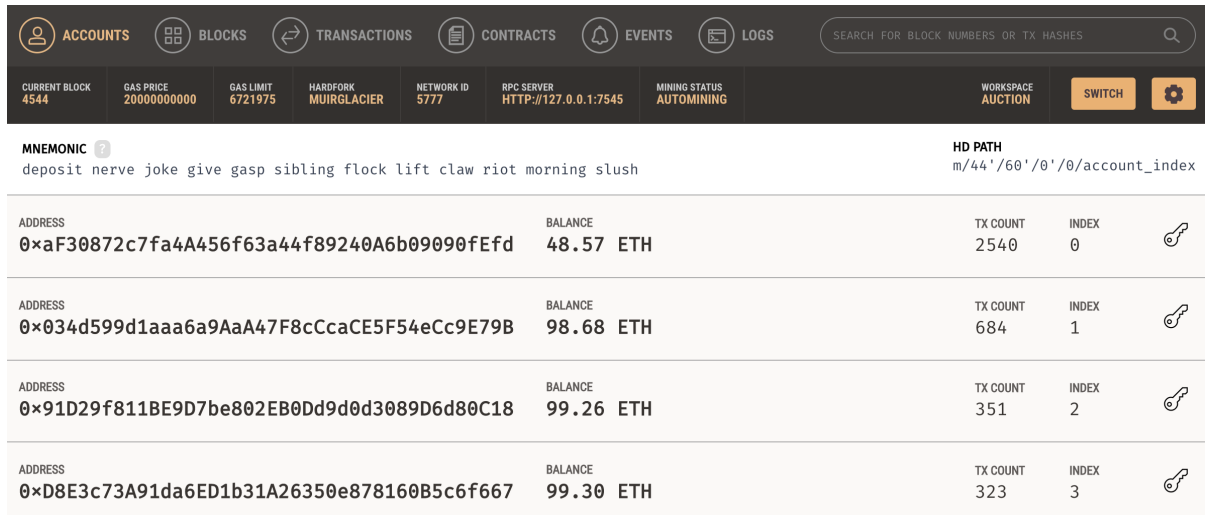
Implementation Assessment

This chapter will demonstrate how the implemented code is tested in order to verify that the system works as expected. The setup and configuration of the test environment is also included. The source code for the Auction Controller tests and the Auction tests can be found in Appendix C and Appendix D respectively, and is also published to Github: <https://github.com/cristinatorp/master-thesis>. Finally, we will conduct a security analysis of the solution with respect to the users' security requirements.

5.1 Setup

Our Solidity tests are written in the Truffle test framework. Truffle offers support for writing tests in both Solidity and Javascript. We decided to write Javascript tests to simulate contacting the contracts like we would do in reality; from a DApp user interface written in JS. This also gives us the possibility to test and verify UI functionality that has not yet been developed. For testing, we use the Ganache network in order to use Ganache's IDE to inspect the blockchain's transactions, logs, events, and accounts (see Figure 5.1). We configure Truffle to use the Ganache network in the `truffle-config.js` file. This is shown in Code Fragment 5.1. In Figure 5.1, you can see the "RPC SERVER" subtitle that denotes "HTTP://127.0.0.1:7545". This means that Ganache runs on localhost (127.0.0.1) on port 7545. The Ganache IDE also lists the network ID: 5777. By listing this information in Truffle's configuration file, running commands such as `truffle test` in the command line will compile and run the test files in the project.

5.1. SETUP



The screenshot shows the Ganache IDE interface. At the top, there are navigation tabs for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS. Below these are various network statistics: CURRENT BLOCK (454), GAS PRICE (2000000000), GAS LIMIT (6721975), HARDFORK (MUIRGLACIER), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), and MINING STATUS (AUTOMINING). The WORKSPACE is set to AUCTION. A search bar is available for block numbers or transaction hashes. Below the statistics, the MNEMONIC is displayed as 'deposit nerve joke give gasp sibling flock lift claw riot morning slush' and the HD PATH is 'm/44'/60'/0'/0/account_index'. The main area shows a table of accounts with their addresses, balances, transaction counts, and indices.

ADDRESS	BALANCE	TX COUNT	INDEX
0xaF30872c7fa4A456f63a44f89240A6b09090fEfd	48.57 ETH	2540	0
0x034d599d1aaa6a9AaA47F8cCcaCE5F54eCc9E79B	98.68 ETH	684	1
0x91D29f811BE9D7be802EB0Dd9d0d3089D6d80C18	99.26 ETH	351	2
0xD8E3c73A91da6ED1b31A26350e878160B5c6f667	99.30 ETH	323	3

Figure 5.1: Ganache IDE: displaying the user accounts, their balance and their tx count

```
1 module.exports = {
2   development: {
3     host: "127.0.0.1",      // Localhost (default: none)
4     port: 7545,           // Ganache port
5     network_id: 5777,     // Ganache network (default: none)
6   },
7 }
```

Code Fragment 5.1: Truffle configuration

Truffle also uses something called *migrations* that deploys contracts to the Ethereum network. As the development continues and the system evolves, new migration scripts can be added. Truffle will only run the migration scripts that have not yet been run. In Code Fragment 5.2 you see an example of how easy it is to deploy a contract to the network with Truffle. Running `truffle migrate` in your console will then give you a result as shown in Figure 5.2.

```
1 let AuctionController = ...
   artifacts.require("./AuctionController.sol");
2
3 module.exports = async (deployer) => {
4   await deployer.deploy(AuctionController);
5 }
```

Code Fragment 5.2: Truffle migration: deploy controller contract

```
> truffle migrate
Compiling your contracts...
> Everything is up to date, there is nothing to compile.

Starting migrations...
> Network name:    'development'
> Network id:     5777
> Block gas limit: 6721975 (0x6691b7)

2_deploy_controller.js

Deploying 'AuctionController'
> transaction hash: 0x67a259a6ad18fbb737463100a8dd6ea4d6ac74c8b29c9c88239cee8ceb1e3210
> Blocks: 0        Seconds: 0
> contract address: 0xCf7038a1C1b61F503766277db89eEf5e2570f9Ec
> block number:    4547
> block timestamp: 1620732837
> account:        0xaF30872c7fa4A456f63a44f89240A6b09090fEfd
> balance:        48.48822504
> gas used:       3779818 (0x39acea)
> gas price:      20 gwei
> value sent:     0 ETH
> total cost:     0.07559636 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:     0.07559636 ETH
```

Figure 5.2: Truffle migrate console results

5.1.1 Testing Third Party Libraries

In addition to Truffle, we have also implemented a couple of third party libraries that makes testing the contracts easier. **Truffle-assertions** [33] include convenience functions for asserting that events are fired when calling contract functions, and to verify the arguments in said events. It also includes functions that assert that a function reverts with the correct error message. This is useful for asserting that the require statements will revert functions when their conditions fail. The **openzeppelin/test-helpers** library [34] includes a handy `time` member that can be used to increase `block.timestamp` with a concrete duration.

5.2 Auction Controller Tests

Unit tests are performed by developers and penetration testers to verify that some part, function, or codeblock of a system behaves as expected. We prepare prerequisite information the unit needs to perform its task, and then inspect the unit's return value with respect to the initial information. Take a function that summarizes two numbers, for

instance. We prepare two variables, $a = 5$ and $b = 10$ and send them in as arguments to the function. The returned value should then be 15, as we know that $5 + 10 = 15$.

Truffle tests require that the contracts' Solidity code files must be imported. `let AuctionController = artifacts.require("./TestAuctionController.sol")` gives us the contract as an object `AuctionController` that can be used in the tests. Code Fragment 5.3 demonstrates how the architecture of the Auction Controller test is set up. The test is injected with the accounts belonging to the configuration, which in our case is the accounts from Ganache. Several global variables are then instantiated, which are constants that will be used throughout the tests. Note that the values of energy amount, minimum bid, and deposit do not reflect what we expect to be real user values, only the variable type. `beforeEach` is a function from Truffle, which will be run before each test. This is where we create the Auction Controller contract which will run the controller's constructor. It is initialized from the owner (admin) account and requires an amount of gas in order for the contract to be deployed.

```
1 contract("AuctionController", accounts => {
2   let contract;
3   const owner = accounts[0];
4   const seller = accounts[1];
5   const energyAmount = 200;
6   const minBid = 5000000;
7   const deposit = 1000000000;
8   const ONE_DAY = 86400;
9
10  beforeEach(async () => {
11    contract = await AuctionController.new(
12      { from: owner, gas: 6700000 }
13    );
14  });
15
16  // Tests here
17 }
```

Code Fragment 5.3: AC test: architecture

The tests for the auction controller comprises verifying that the contract is initialized in the expected fashion, that the controller can deploy new auction contracts, that only the admin can delete auctions, and that the auction's cannot be deleted prematurely.

Each truffle test is denoted with the syntax `it("test name")`. Code Fragment 5.4 presents the test for deploying new auctions through the Auction Controller (AC). It calls the controller's `deployNewAuction` function with the seller, energy amount, minimum bid value, and deposit as arguments. The auction address can be found in that transaction's logs. If the function works as expected, the seller's address should be saved in the AC's `sellerAddresses` mapping, and we can access this by calling said mapping with the auction address as the key. We then verify that the seller address stored on the contract is the same seller address that we passed as an argument in deployment. This is done with Truffle's `expect` function: `expect(<expectedValue>).to.equal(actualValue)`.

```
1 it("can deploy new auction contract", async () => {
2     const tx = await contract.deployNewAuction(
3         seller,
4         energyAmount,
5         minBid,
6         deposit,
7     );
8
9     const newAuctionAddress = tx.logs[0].args.auction;
10    const sellerAddress = await ...
11        contract.sellerAddresses.call(newAuctionAddress);
12    expect(sellerAddress).to.equal(seller);
13
14    const auction = await Auction.at(newAuctionAddress);
15    const {1: aSeller, 2: aEnergyAmount, 3: aMinBid, 4: ...
16        aDeposit} = await auction.getAuctionInfo();
17    expect(aSeller).to.equal(seller);
18    expect(Number(aEnergyAmount)).to.equal(energyAmount);
19    expect(Number(aMinBid)).to.equal(minBid);
20    expect(Number(aDeposit)).to.equal(deposit);
21    truffleAssert.eventEmitted(tx, "AddedNewAuction");
22 }
```

Code Fragment 5.4: AC test: can deploy new auction

We can retrieve the auction contract instance by importing the Auction code with `artifacts.require("TestAuction.sol")`. The auction instance can then be accessed with `Auction.at(<address>)`, and we retrieve the auction info stored in the `AuctionInfo`

struct. Again, we verify that the seller's Ethereum account address, energy amount, minimum bid value, and deposit value that we passed as arguments when deploying the auction has been stored correctly in the auction contract with expect statements. Note that we convert the `uint256` values retrieved from the auction to a regular number, as these are stored as Big Numbers and will not pass equality checks when compared to the global variables set in the tests. Finally, we verify that the `AddedNewAuction` event was emitted from the AC with the help of the `truffle-assertions` library. If any of the `expect` or `eventEmitted` functions fail during the test, so will the test.

The next test verifies that an Ethereum account that is not admin or the seller of the specific auction cannot delete the auction (Code Fragment 5.5). We deploy the auction and obtain the auction address in the same fashion as in the last test. Then, we use the `reverts` function of the Truffle Assertions library to verify that the AC's function `deleteAuction()` fails when we call it from the third Ganache account (recall that the first account is the admin, and the second account is the seller). The function should revert with the error message "Can only be deleted by admin or the auction seller". This test passes if the function reverts, and fails if the account is allowed to delete the auction.

```
1 it("cannot delete auction if not admin or auction seller", ...
  async () => {
2     const tx = await contract.deployNewAuction(
3         seller,
4         energyAmount,
5         minBid,
6         deposit,
7     );
8
9     await truffleAssert.reverts(
10        contract.deleteAuction(tx.logs[0].args.auction,
11        { from: accounts[2] }),
12        "Can only be deleted by admin or the auction seller"
13    );
14 });
```

Code Fragment 5.5: AC test: can not delete if not admin

In Code Fragment 5.6, we verify that even the admin is not allowed to delete an auction if the auction is not finished. This test specifically tests that the auction cannot be deleted while in the hidden round. In Appendix C we have included similar tests for validating that the auction cannot be deleted in any other states as well. The auction must be in the state `ReadyForDeletion`, and the token must either be retrieved already, or be outdated. The test tries to delete the auction right after it has been created from the admin account, and it expects the function to revert with the error message "Cannot delete auction before the token has expired or been retrieved".

```
1 it("cannot delete auction if in the hidden round", async () => {
2     const tx = await contract.deployNewAuction(
3         seller,
4         energyAmount,
5         minBid,
6         deposit,
7     );
8
9     await truffleAssert.reverts(
10        contract.deleteAuction(tx.logs[0].args.auction,
11        { from: owner }),
12        "Cannot delete auction before the token has expired or ...
13        been retrieved"
14    );
15 });
```

Code Fragment 5.6: AC test: cannot delete auction prematurely

The last test we will present in this section is to verify that the admin can delete the auction if the auction has been completed. In Appendix C we have also included a test that verifies that the seller can delete his own auction, and another that verifies that the seller cannot delete another seller's auction. In Code Fragment 5.7, we deploy a new auction and obtains its address which we use to access the auction instance. We then bid in the hidden round (note that we use web3's `soliditySha3` hashing function to hash the bid with a salt), speed time up by a day, close the hidden round, bid in the open round (where the bid and salt used is sent in clear text), speed time up by another day, close the auction, and finally retrieve the token from the winner account address. We know that the third account is the winner account, as this is the only account we submitted a bid from in the test. Finally, we delete the auction from the admin account (owner)

and verify that the event `DeletedAuction` was emitted, and logged the correct auction address.

```
1 it("admin can delete auction", async () => {
2     const tx = await contract.deployNewAuction(
3         seller,
4         energyAmount,
5         minBid,
6         deposit,
7     );
8
9     const newAuctionAddress = tx.logs[0].args.auction;
10    const auction = await Auction.at(newAuctionAddress);
11
12    await auction.bidInHiddenRound(
13        web3.utils.soliditySha3(minBid, "some_salt"),
14        { from: accounts[2], value: deposit }
15    );
16    await time.increase(ONE_DAY + 1);
17    await auction.closeHiddenRound();
18    await auction.bidInOpenRound(minBid, "some_salt", { from: ...
19        accounts[2] });
20    await time.increase(ONE_DAY + 1);
21    await auction.closeAuction();
22    await auction.retrieveToken({ from: accounts[2] });
23
24    const deleteTx = await ...
25        contract.deleteAuction(newAuctionAddress, { from: owner });
26    truffleAssert.eventEmitted(deleteTx, "DeletedAuction",
27        (ev) => ev.auction == newAuctionAddress);
28    });
```

Code Fragment 5.7: AC test: admin can delete auction

When using `truffle test` to run test files, it is possible to specify a single test file to only run the tests in said file. In Figure 5.3, we present the log that will be outputted in the console when only running the tests for the Auction Controller. It outputs the name of the tests, a green tick mark for all passed tests, and the time spent for each test. If any of the tests failed during testing, it would log all events that were emitted during

the failed function(s) and print the error message(s). In our case, all nine tests passed. The complete source code of this file is found in Appendix C.

```
> truffle test ./test/auction_controller_test.js
Using network 'development'.

Compiling your contracts...
=====
> Compiling ./contracts/Auction.sol
> Compiling ./contracts/AuctionController.sol
> Compiling ./contracts/TestAuction.sol
> Compiling ./contracts/TestAuctionController.sol
> Artifacts written to /var/folders/mh/446d2mds2zscpkt_1zfbmkl80000gn/T/test--24752-0Qj11CPYRbEs
> Compiled successfully using:
   - solc: 0.8.1+commit.df193b15.Emscripten.clang

Contract: AuctionController
  ✓ contract is initialized
  ✓ can deploy new auction contract (266ms)
  ✓ cannot delete auction if not admin or auction seller (1120ms)
  ✓ cannot delete auction if in the hidden round (275ms)
  ✓ cannot delete auction if in the open round (431ms)
  ✓ cannot delete auction if the token has not yet expired or been retrieved (606ms)
  ✓ admin can delete auction (692ms)
  ✓ seller can delete auction (570ms)
  ✓ seller can delete his own auction, but not one from another seller (1178ms)

9 passing (6s)
```

Figure 5.3: All auction controller tests passed

5.3 Auction Tests

Due to the fact that the Auction contract handles more functionality than the Auction Controller contract, there are a significant amount of tests for this contract. We will verify that the contract is initialized as expected, that a bidder can bid in the hidden round and in the open round, that the auction closes the bidding rounds at the appropriate times, that the auction finds the correct winner and creates a token for that account, that the auction transfers back deposits to all valid bidders and the highest bid to the seller, and that the winner (and only the winner) can retrieve its token. These are structured into three categories: (1) Tests During Hidden Round; (2) Test During Open Round; and (3) Tests For Closing The Auction.

In order to test aspects of our auction, we must create another contract `TestAuction` (Code Fragment 5.8). Although we can modify the current time with OpenZeppelin’s test-helper library, we cannot modify the current state of the Auction with such a third party library. The `TestAuction` contract (child) is set to *inherit* the `Auction` contract (parent),

which means that the child can access all non-private members of the parent. It can also access *internal* functions in the parent. In `TestAuction`, we can therefore modify the current state through `setCurrentState(State newState)`, and call the internal functions `findWinner()`, `transferBackDeposits()`, and `transferHighestBidToSeller()` which normally cannot be accessed externally.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.1;
3 import "./Auction.sol";
4
5 contract TestAuction is Auction {
6     constructor(
7         address payable _seller,
8         uint _energyAmount,
9         uint _minBidValue,
10        uint _depositValue
11    ) Auction (
12        _seller,
13        _energyAmount,
14        _minBidValue,
15        _depositValue
16    ) {}
17
18    function setCurrentState(State newState) public {
19        auctionInfo.currentState = newState;
20    }
21
22    function testFindWinner() public {
23        findWinner();
24    }
25
26    function testTransferBackDeposits() public {
27        transferBackDeposits();
28    }
29
30    function testTransferHighestBidToSeller() public {
31        transferHighestBidToSeller();
32    }
33 }
```

Code Fragment 5.8: Test Auction, a helper contract for testing

Code Fragment 5.9 presents the global constant variables used throughout the auction tests. We set the first Ganache account to be the seller, declare the four states that the auction can be in with integers from 0 to 3, and a constant variable for one day in seconds (86000). The test values for the energy amount, minimum bid value, and deposit value for the auction is set to arbitrary values. Then we create an array `MOCK_BIDS` which contains five different values that will be used to simulate bids, and finally, create an auction instance in Truffle's `beforeEach()` function.

```
1 contract("Auction", accounts => {
2   let contract;
3   const sellerAccount = accounts[0];
4
5   const READY_FOR_HIDDEN_BIDS_STATE = 0;
6   const READY_FOR_OPEN_BIDS_STATE = 1;
7   const CLOSED_STATE = 2;
8   const READY_FOR_DELETION_STATE = 3;
9
10  const ONE_DAY = 86400;
11  const ENERGY_AMOUNT = 200;
12  const MIN_BID_VALUE = 50000;
13  const DEPOSIT_VALUE = 100000;
14
15  const MOCK_BIDS = [MIN_BID_VALUE + 2, MIN_BID_VALUE + 4, ...
16    MIN_BID_VALUE + 1, MIN_BID_VALUE + 3, MIN_BID_VALUE];
17
18  beforeEach(async () => {
19    contract = await Auction.new(
20      sellerAccount,
21      ENERGY_AMOUNT,
22      MIN_BID_VALUE,
23      DEPOSIT_VALUE,
24      { gas: 4000000 }
25    );
26  });
27
28  // Tests here
29 }
```

Code Fragment 5.9: Auction test: architecture

5.3.1 Tests During Hidden Round

The first test we run is to verify that the auction was initialized correctly (Code Fragment 5.10). We call the `getAuctionInfo()` function from the Auction contract, store the returned information in the object variable `a` and compare each value to the constant global variables. The current state should be initialized to 0 (ReadyForHiddenBids), the seller's account address, energy amount, minimum bid value, and deposit should be initialized to the variables passed as arguments in deployment, and the two deadlines should be initialized to one day (86400 seconds) and two days (172800 seconds) respectively.

```

1  it("contract is initialized", async () => {
2      let a = await getAuctionInfo();
3      const latestTime = await time.latest();
4
5      expect(a.currentState).to.equal(READY_FOR_HIDDEN_BIDS_STATE);
6      expect(a.seller).to.equal(sellerAccount);
7      expect(a.energyAmount).to.equal(ENERGY_AMOUNT);
8      expect(a.minBidValue).to.equal(MIN_BID_VALUE);
9      expect(a.depositValue).to.equal(DEPOSIT_VALUE);
10     expect(a.hiddenBidsDeadline).to.equal(latestTime.toNumber() ...
        + ONE_DAY);
11     expect(a.openBidsDeadline).to.equal(latestTime.toNumber() + ...
        (ONE_DAY * 2));
12 });
13
14 getAuctionInfo = async () => {
15     let info = await contract.getAuctionInfo.call();
16     return {
17         "currentState": Number(info[0]),
18         "seller": info[1],
19         "energyAmount": Number(info[2]),
20         "minBidValue": Number(info[3]),
21         "depositValue": Number(info[4]),
22         "hiddenBidsDeadline": Number(info[5]),
23         "openBidsDeadline": Number(info[6]),
24     };
25 };

```

Code Fragment 5.10: Auction test: initialization

In Code Fragment 5.11, we assert that a bidder can bid in the hidden round. We call the contract's `bidInHiddenRound()` function and pass the bid value hashed with `soliditySha3` along with the salt `"some_salt"` as the function's single argument. We set the transaction value to be the deposit value and send it from the second Ganache account. Now, we can retrieve the stored bid by accessing it from the mapping `bids` with the seller's account address as the key. This will return a `Bid` struct, which will be converted to an object in JavaScript. The test will pass if the bid's `existsHiddenBid` bool is equal to `true`, and if its deposit value is equal to the actual deposit value we transferred with the bid transaction.

```
1 it("can bid in hidden round", async () => {
2   await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
3     DEPOSIT_VALUE);
4   const bid = await contract.bids.call(accounts[1]);
5   expect(bid.existsHiddenBid).to.equal(true);
6   expect(Number(bid.deposit)).to.equal(DEPOSIT_VALUE);
7 });
8
9 bidInHiddenRound = async (bidValue, bidderAddress, ...
10   depositValue) => {
11   let tx = await ...
12     contract.bidInHiddenRound(web3.utils.soliditySha3(bidValue, ...
13       "some_salt"), {
14         value: depositValue,
15         from: bidderAddress
16       });
17   truffleAssert.eventEmitted(tx, "ReceivedHiddenBid", (ev) => {
18     return ev.bidder == bidderAddress && ev.deposit == ...
19       depositValue;
20   });
21 }
```

Code Fragment 5.11: Auction test: can bid in hidden round

Several tests are included to verify possible situations where the bidder is not allowed to bid in the hidden round. All of them can be found in Appendix D. Code Fragment 5.12 presents a test that asserts that bids should not be accepted if the deposit value is lower

than the minimum deposit value selected by the seller. It uses the `reverts()` function of the `truffle-assertions` library and verifies that the function reverts with the error message "Deposit value is too low". Code Fragment 5.13 presents another test where the deposit value is valid, but the bid is attempted to be submitted after the hidden bids deadline has expired. This case is simulated by increasing the time by one day + 1 second (86001 seconds) with the help of OpenZeppelin's test-helper library. This test will pass if the function reverts with the error message "Cannot bid after deadline".

```
1 it("cannot bid in hidden round if deposit is too low", async () ...
  => {
2   await truffleAssert.reverts(
3     bidInHiddenRound(
4       MIN_BID_VALUE,
5       accounts[1],
6       DEPOSIT_VALUE - 1
7     ),
8     "Deposit value is too low"
9   );
10 });
```

Code Fragment 5.12: Auction test: cannot bid if deposit is too low

```
1 it("cannot bid in hidden round if after deadline", async () => {
2   await time.increase(ONE_DAY + 1);
3   await truffleAssert.reverts(
4     bidInHiddenRound(
5       MIN_BID_VALUE,
6       accounts[1],
7       DEPOSIT_VALUE
8     ),
9     "Cannot bid after deadline"
10  );
11 });
```

Code Fragment 5.13: Auction test: cannot bid in hidden round after hidden bids deadline

Another case that is important to test is that the auction closes when no bids were transferred during the hidden round. This test is presented in Code Fragment 5.14. We prepare the test by increasing the time to after the hidden bids deadline and then close

the hidden round. We assert that the hidden round was closed by verifying that the event `ClosedAuctionWithNoBids` was emitted with the log argument `whichRound` equal to `"Hidden round"`. As we have not simulated any bids during this test, the length of the contract's array `hiddenBidsAddresses` should be equal to 0. The final test is to verify that the current state of the auction has advanced to `ReadyForDeletion` when no bids were received during the hidden round.

```

1 it("auction should close if no hidden bids were received", ...
  async () => {
2   await time.increase(ONE_DAY + 1);
3
4   const tx = await contract.closeHiddenRound();
5   truffleAssert.eventEmitted(tx, "ClosedAuctionWithNoBids", ...
      (ev) => {
6     return ev.whichRound == "Hidden round";
7   });
8
9   const hiddenBidsLength = Number(await ...
      contract.getHiddenBidsLength());
10  expect(hiddenBidsLength).to.equal(0);
11
12  const state = Number(await contract.getCurrentState());
13  expect(state).to.equal(READY_FOR_DELETION_STATE);
14 });

```

Code Fragment 5.14: Auction test: close auction if no hidden bids were received

There are also tests that assert that the auction advances to the open bid round when valid bids were received and that the `closeHiddenRound()` function cannot be called if the current state is not `ReadyForHiddenBids` or if the hidden bids deadline has not yet expired. These are included in Appendix C.

5.3.2 Tests During Open Round

Similar to the test that verifies that a bidder can bid in the hidden round, a test is also written that verifies that a bidder can bid in the open round. This is presented in Code Fragment 5.15. We prepare the test by simulating a bid in the hidden round, increasing the time by one day, and closing the hidden round. Then we call the auction's `bidInOpenRound()` function and pass the same bid value and salt which we gave during

the hidden round, this time as open arguments in plaintext. It is also important to bid from the same account as was simulated in the hidden round. We then retrieve the bid from the `Bid` struct by accessing it with the bidder's account address as the key. The test passes if the bid's `isOpenBidValid` member equals `true` and `openBid` equals the bid value we passed as an argument in `bidInOpenRound()`.

```
1 it("can bid in open round", async () => {
2   await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
   DEPOSIT_VALUE);
3   await time.increase(ONE_DAY + 1);
4   await contract.closeHiddenRound();
5   await contract.bidInOpenRound(MIN_BID_VALUE, "some_salt", { ...
   from: accounts[1] });
6
7   let bid = await contract.bids.call(accounts[1]);
8
9   expect(bid.isOpenBidValid).to.equal(true);
10  expect(Number(bid.openBid)).to.equal(MIN_BID_VALUE);
11 });
```

Code Fragment 5.15: Auction test: can bid in open round

There are also tests that verify that a bidder cannot bid in the open round if the auction's current state is not `ReadyForOpenBids`, if the open bids deadline has expired, or if the bid value is below the minimum bid value set by the seller. These are omitted for brevity.

The next test we will present is verifying that a bidder is not allowed to bid in the open round if he did not participate in the hidden round (Code Fragment 5.16). We begin by simulating a hidden bid from the second Ganache account, increase the time by one day and then close the hidden round. Note that we must simulate a hidden bid here to prevent that the auction closes with no bids. We then try to bid in the open round from the third Ganache account. We expect that the function should revert with the error message "This account has not bidden in the hidden round". If the attempt to bid in the open round from the third account does not fail, so does the test.

```
1 it("cannot bid in open round if not already bidden in hidden ...
  round", async () => {
2     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
      DEPOSIT_VALUE);
3     await time.increase(ONE_DAY + 1);
4     await contract.closeHiddenRound();
5
6     await truffleAssert.reverts(
7         contract.bidInOpenRound(MIN_BID_VALUE, "some_salt", { ...
          from: accounts[2] }),
8         "This account has not bidden in the hidden round"
9     );
10 });
```

Code Fragment 5.16: Auction test: cannot bid in open round if bidder did not participate in hidden round

Another case that is important to test is that the hidden bid should match the open bid. This is presented in Code Fragment 5.17. We prepare the test in the same way as in the previous case, except that we send both bids from the same account. Additionally, we alter the bid sent in the open round ($\text{MIN_BID_VALUE} + 1$ instead of MIN_BID_VALUE).

```
1 it("cannot bid in open round if bid does not match hidden bid", ...
  async () => {
2     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
      DEPOSIT_VALUE);
3     await time.increase(ONE_DAY + 1);
4     await contract.closeHiddenRound();
5
6     await truffleAssert.reverts(
7         contract.bidInOpenRound(MIN_BID_VALUE + 1, "some_salt", ...
          { from: accounts[1] }),
8         "Open bid and hidden bid do not match"
9     );
10 });
```

Code Fragment 5.17: Auction test: cannot bid in open round if the open bid does not match the hidden bid

Recall that in the auction's `bidInOpenRound()` function, the bid passed as its argument will be hashed along with the given salt and compared with the bidder's hidden bid. Because the hidden bid and the open bid differ, this will result in two different hash strings, and the function will revert with the error message "Open bid and hidden bid do not match".

The last test we will include in this subsection is closing the open round successfully (Code Fragment 5.18). We simulate the bidding process by bidding in the hidden and open round with four different Ganache accounts. This simulation is represented by Code Fragment 5.19. We pass the array `MOCK_BIDS` which contains five values: `MIN_BID_VALUE + 2`, `MIN_BID_VALUE + 4`, `MIN_BID_VALUE + 1`, `MIN_BID_VALUE + 3`, and `MIN_BID_VALUE`. For each bid i , we simulate a hidden bid from different Ganache accounts (second to sixth account, specifically). Simply by inspecting these bids, we can see that the second bid is the highest (`MIN_BID_VALUE + 4`). This is sent from the third Ganache account. We validate that the contract has received and stored five different bids by checking the length of its `hiddenBidsAddresses` array. We then increase the time by one day, close the hidden round, and bid once more in the open round. We use the same bids from the same accounts, and these should all be valid. The highest bid and the account that submitted this bid will be returned from the `mockBidding()` function.

```

1 it("closed open round", async () => {
2   await mockBidding(MOCK_BIDS);
3   await time.increase(ONE_DAY + 1);
4   const tx = await contract.closeOpenRound();
5
6   let a = await getAuctionInfo();
7   expect(a.currentState).to.equal(CLOSED_STATE);
8   truffleAssert.eventEmitted(tx, "ClosedRound", (ev) => ...
9     ev.whichRound == "Open round");
  });

```

Code Fragment 5.18: Auction test: close open round successfully

When all bids are simulated, we increase the time again by one day and close the open round. This is a function that is added only for testing purposes, as we usually call `closeAuction()`, not `closeOpenRound()`. The difference is that the former finds the winner and performs all the closing logic as described in detail in subsection 4.2.6. The latter closes the open round and advances to the next phase if it did receive valid open

bids during the open round. Considering that we submitted five valid bids, we assert that the auction's current state is `Closed`, and that the event `ClosedRound` was emitted with the argument `whichRound` equal to "Open round".

```
1 mockBidding = async (bids, includeInvalidBid = false) => {
2   for (let i = 0; i < bids.length; i++) {
3     await bidInHiddenRound(
4       bids[i],
5       accounts[i + 1],
6       DEPOSIT_VALUE
7     );
8   }
9   let hiddenBidsNum = Number(await ...
10    contract.getHiddenBidsLength.call());
11   expect(hiddenBidsNum).to.equal(bids.length);
12
13   await time.increase(ONE_DAY + 1);
14   await contract.closeHiddenRound();
15
16   for (let i = 0; i < bids.length; i++) {
17     if (includeInvalidBid && i == 0) {
18       await truffleAssert.reverts(
19         bidInOpenRound(
20           bids[i] - 1,
21           "some_salt",
22           accounts[i + 1]
23         ),
24         "Open bid and hidden bid do not match"
25       );
26       continue;
27     }
28     await bidInOpenRound(bids[i], "some_salt",
29       accounts[i + 1]);
30
31   }
32
33   let highestBid = Math.max(...bids);
34   let highestBidder = accounts[bids.indexOf(highestBid) + 1];
35   return { "bid": highestBid, "bidder": highestBidder};
36 };
```

Code Fragment 5.19: Auction test: simulate bidding with different accounts

The remaining tests for the open round comprises verifying that the auction should close and advance to `ReadyForDeletion` if it did not receive any valid open bids, and that we cannot close the open round if the current state is not `ReadyForOpenBids` or if the open bids deadline has not expired yet. These are omitted for brevity.

5.3.3 Tests For Closing The Auction

The first test we will present in this phase is finding the correct winner of an auction (Code Fragment 5.20). As we described in the previous subsection, the convenience function `mockBidding()` returns an object that consists of the bid and the account address of the highest bid of the simulated bids in `MOCK_BIDS`. We increase the time by one day and close the open round. We then call the `findWinner()` function independently. Recall that this function is *internal*, and we implemented the Test Auction contract to access the internal functions in the Auction contract. We then retrieve the stored winner by calling the contract's `Winner` struct. Now we can compare the winner's account address with the account address we know submitted the highest bid and the winner's bid with the highest bid in `MOCK_BIDS`. If these two tests pass, and the `FoundHighestBid` event was emitted, the correct winner has been found.

```
1 it("found auction winner", async () => {
2   const actualHighestBid = await mockBidding(MOCK_BIDS);
3   await time.increase(ONE_DAY + 1);
4   await contract.closeOpenRound();
5   const tx = await contract.testFindWinner();
6
7   const winner = await contract.winner.call();
8
9   truffleAssert.eventEmitted(tx, "FoundHighestBid");
10  expect(winner.accountAddress).to.equal(actualHighestBid.bidder);
11  expect(Number(winner.bid)).to.equal(actualHighestBid.bid);
12 });
```

Code Fragment 5.20: Auction test: finding the correct winner

We will also demonstrate that deposits will not be transferred back to bidders that made a fraudulent bid attempt (Code Fragment 5.21). We simulate this test case by declaring a variable `invalidBidder` which stores the second Ganache account. We then run `mockBidding()` again, but this time we set the optional parameter `includeInvalidBid`

to true. When this parameter is set to true, the bid for the invalid bidder account will be modified during the open bid round (see Code Fragment 5.19). We expect the `bidInOpenRound()` function to revert with the error message "Open bid and hidden bid do not match", and then continue with the rest of the bids. Then, we prepare the test further by setting the auction's state to `Closed` and find the auction winner. We will now retrieve the balance of the invalid bidder's account. This will be our `balanceBefore` variable. Then, we call the contract's `transferBackDeposits()` function and verify that a `TransferEvent` was emitted. When that is finished, we again retrieve the invalid bidder's account balance and store it in `balanceAfter`. If the contract did not transfer Ether to the invalid bidder's account during this function, the difference between these two balances should be 0.

```
1 it("did not send deposit back to invalid bidder", async () => {
2   const invalidBidder = accounts[1];
3   await mockBidding(MOCK_BIDS, true); // Include invalid ...
      first bid
4   await contract.setCurrentState(CLOSED_STATE);
5   await contract.testFindWinner();
6
7   let balanceBefore = await getBalance(invalidBidder);
8   let tx = await contract.testTransferBackDeposits();
9   truffleAssert.eventEmitted(tx, "TransferEvent");
10  let balanceAfter = await getBalance(invalidBidder);
11
12  expect(Number(balanceAfter - balanceBefore)).to.equal(0);
13 });
```

Code Fragment 5.21: Auction test: did not transfer deposit back to invalid bidder

Next, we will assert that the seller receives his payment as well as any extra deposits that may remain on the contract from invalid bidders. This is shown in Code Fragment 5.22. We prepare the test in a similar manner to the previous test, including an invalid bid in `mockBidding()`, closing the open round, finding the winner, and transferring back the deposits to all valid bidders. Then we retrieve the seller's account balance in `balanceBefore` and run the contract's `transferHighestBidToSeller()` function. We verify that a `TransferEvent` was emitted, and then retrieve the seller's account balance again after the payment should be transferred. Subtracting the `balanceBefore` value

from the `balanceAfter` value should be equal to the highest bid + one deposit, as we only included one invalid bid.

```
1 it("sent highest bid to seller, one extra deposit", async () => {
2   const highestBid = await mockBidding(MOCK_BIDS, true); // ...
3   Include invalid first bid
4   await time.increase(ONE_DAY + 1);
5   await contract.closeOpenRound();
6   await contract.testFindWinner();
7   await contract.testTransferBackDeposits();
8
9   const balanceBefore = BigInt(await ...
10    web3.eth.getBalance(sellerAccount));
11   const tx = await contract.testTransferHighestBidToSeller({ ...
12    gasPrice: 0});
13   truffleAssert.eventEmitted(tx, "TransferEvent");
14   const balanceAfter = BigInt(await ...
15    web3.eth.getBalance(sellerAccount));
16
17   expect(Number(balanceAfter - ...
18    balanceBefore)).to.equal(highestBid.bid + DEPOSIT_VALUE);
19 });
```

Code Fragment 5.22: Auction test: sent highest bid to seller, as well as any remaining deposits

```
1 it("winner retrieved token", async () => {
2   await mockBidding(MOCK_BIDS);
3   await time.increase(ONE_DAY + 1);
4   await contract.closeAuction();
5   const winner = await contract.winner.call();
6
7   const tx = await contract.retrieveToken(
8     { from: winner.accountAddress }
9   );
10  truffleAssert.eventEmitted(tx, "RetrievedToken");
11 });
```

Code Fragment 5.23: Auction test: winner can retrieve token

Code Fragment 5.23 presents a test that demonstrates that the auction winner is allowed to retrieve the token generated by the auction. We simulate the bidding rounds without invalid bids this time, increase the time by one day, close the auction, and retrieve the winner by calling the `Winner` struct. Then, we call the contract's `retrieveToken()` function from the winner's Ganache account. If the event `RetrievedToken` is emitted, the function was run without errors.

It is, however, also important to verify that an arbitrary account that is not the auction winner is not allowed to retrieve the auction token. This is demonstrated in Code Fragment 5.24. Here, we call the contract's `retrieveToken()` function again, but from the second Ganache account (recall that the winner is the third Ganache account). This test will pass if the function reverts with the error message "You are not the winner of the auction!".

```
1 it("non-winner is not allowed to retrieve token", async() => {
2   await mockBidding(MOCK_BIDS);
3   await time.increase(ONE_DAY + 1);
4   await contract.closeAuction();
5
6   await truffleAssert.reverts(
7     contract.retrieveToken({ from: accounts[1] }),
8     "You are not the winner of the auction!"
9   );
10 });
```

Code Fragment 5.24: Auction test: only the auction winner can retrieve the token

The last test we will include in this chapter is to assert that the auction contract's private variable `token` is not available to the public, and is presented in Code Fragment 5.25. The test is prepared by simulating the bidding rounds, increasing the time by one day, and closing the auction. Then, we try to call the token variable directly. We wrap this statement in a try/catch statement, as we expect it to fail. When the function throws the error, we catch it and verify its error message. This should be "Cannot read property 'call' of undefined", as the `token` variable is hidden on the blockchain and will not be listed in the auction contract's state variables.

```
1 it("token should not be callable", async () => {
2     await mockBidding(MOCK_BIDS);
3     await time.increase(ONE_DAY + 1);
4     await contract.closeAuction();
5
6     try {
7         await contract.token.call();
8         expect.fail();
9     } catch(error) {
10        expect(error.message).to.equal("Cannot read property ...
11            'call' of undefined");
12    }
13 });
```

Code Fragment 5.25: Auction test: token is not callable

5.4 Security Analysis

As a part of the assessment of our solution, some security aspects should be addressed. In this subsection, we will discuss how our design utilizes some of the core functionality of blockchain to provide security for our smart contracts by identifying plausible attacks against the solution and how they can be addressed. Security has been a high priority during the thesis and has been a priority under the entire process.

5.4.1 Attack Vectors

In this section, we have provided evidence for the security of the solution via discussing how the system can withstand denial of service attacks, replay attacks, man-in-the middle attacks as well as brute-force attacks. Some of these attacks may be combined or chained together. This would, however, not increase the potential threat given that the proposed solution is resistant against each of the aforementioned threats.

Denial of Service Denial of service (DoS) is an attack that targets availability. Two attack vectors are relevant for the proposed solution: attacks that target the Ethereum network and attacks against the libraries and wallet applications. The Ethereum network is very resilient to denial of service attacks [19], and prolonged attacking with any degree of denial of service will cost the attacker a large amount of Ether. The attacker would

need to spend a large amount of Ether to make the auction last a small amount of time longer, and normal users would barely notice a difference. An attack against wallet applications or libraries would, however, be a more realistic scenario. Similar attacks have previously led to a financial loss for users [35]. The smart contracts do not utilize wallets or libraries. Therefore the auction would function as designed if these were targeted and brought down. The only consequence for the users would be that they will be unable to join a new auction.

Replay attack Replay attacks against the solution would attempt to target either the deposit transaction or the energy token. The auction contract is created and deleted in each auction, and the entire auction transpires on the smart auction contract. This means that our solution is not vulnerable to replay attacks that target design faults in solutions that utilize proxy contracts [35]. A replay attack that attempts to replay the deposit transaction can choose two timeframes for the attack: while the transaction is sent but not yet on the blockchain, and when the transaction is completed and is on the blockchain. If the attacker replays the deposit message before it is on the blockchain, the contract will only deposit once because the other request will be invalid. If the attacker attempts to replay the attack after it is on the chain, the request would be invalid, and it would be futile either way due to the fact that the auction only allows each bidder to bid once. The energy token could use a similar security measure that states that the token is linked to a particular user when the token is sent to the winner. When the winner spends their token, it is no longer valid. A replay attack targeting the token retrieval or the token spending would then be invalid.

Man-in-the-Middle attacks are most often used for information gathering or to further facilitate other attacks. If the attacker manages to perform a man-in-the-middle attack on our solution, they would be stationed between the users and the platform. The information-gathering the attacker has access to there would not be a threat due to the assumed secure connection between the user and the DApp User Interface, and replay attacks were addressed in the previous paragraph. Another possible attack vector mounted from the man-in-the-middle connection would be an attempt to manipulate the data in transit. The data in transit between the users and the platform is the transactions. These transactions will be rejected without the matching signature. The attacker could attempt to perform a **digital signature forgery** to forge a valid signature. If they managed to forge the valid signature of the user, then the attack could be used to break

the security that is provided by a digital signature scheme. A digital signature forgery attack against our solution would consist of a *known signatures attack*, due to the public key and signature of the legal signer would be accessible to the attacker. *Existential Forgery* is when the adversary successfully manages to forge the signature of one message; this message could be any message from the target [24]. Ethereum signatures use ECDSA digital signature scheme. Given that ECDSA algorithms are secure against existential forgery [24], it makes it practically infeasible to create a valid ECDSA signature without having access to the relevant user's private key.

Brute-force In the proposed solution, the bids are sent as hashed messages. This means that if an attacker wants to reveal a bid in an auction, they would have to break the hash. This would make such an attempt a brute-force attack since the only way to break this hash is through brute force. The hashing algorithm in the solution is Keccak-256. Keccak-256 is a one-way collision-resistant hash function [24], combined with the timeline of the hidden round being 24 hours, making a brute-force attack on our solution practically infeasible.

Chapter 6

Conclusion

This chapter concludes the thesis by outlining the case we addressed, summarizing our main findings, and discussing how we have achieved the research objectives presented in Chapter 1. Finally, we have discussed some possible improvements that can be integrated into the solution as a part of future work.

The energy trading market has largely been a monopoly of regional energy providers. Utilizing blockchain technology on the smart grid leads to the decentralization of said market and gives the market control back to the consumers and prosumers. The blockchain also offers benefits such as automation, security, transparency, and immutability, which are properties we appreciate in an eAuction platform. The case we addressed was trading energy in a microgrid where prosumers in a specific neighborhood can auction off excess energy obtained through renewable energy sources (i.e. solar panels). Consumers can then submit bids for this energy in an auction. Due to mutual distrust between bidders, we had to design a secure model for a blind auction, where bids are hidden. In order to decentralize the trading platform and thus removing the need for regional energy providers and intermediaries, we designed this platform with the help of the Ethereum blockchain. As the blockchain is inherently transparent, creating a blind auction where the bids must be hidden created an interesting problem. We also had to consider how the bidders could validate that they had enough funds to transfer the payment transaction without revealing their actual bids in order to avoid fraudulent bid attempts.

In our proposed solution, we have implemented a Decentralized Application (DApp) that resides in the blockchain. Our two user types, the seller and the bidder(s) interact with the DApp throughout an auction life cycle. In order to solve the two main problems described above, we designed our solution to include two separate bidding rounds. The first round is the hidden round, where the bids are hashed along with a salt and sent as data in a deposit transaction to the DApp. Requiring that bidders transfer a deposit during the first round will ensure that the seller receives his payment when the auction closes, as the auction winner's bid will be subtracted from the winner's deposit. The second round is where the bidders reveal their bids. They send their bid along with the salt in plaintext to the DApp, which will compare the bid with the bidders' respective hidden bid. The DApp is responsible for authenticating the bidders via their Ethereum account address and their transaction signatures, validating bid and deposit values, and performing the logic of closing an auction. As this solution model is capable of satisfying the transparency and bidder's balance verification problem, this suggests that we have fulfilled Research Objective **RO 1**.

Our implementation of the solution separated the DApp into three entities: the DApp user interface (UI) that the users interact with, the smart contract `AuctionController` that manages all auction instances, and the smart contract `Auction` that manages a specific auction instance. These contracts were deployed to Ethereum's test network Ganache, while the UI was considered out of scope. The implementation describes how our solution was translated to code using Solidity, which is the primary language for writing smart contracts for Ethereum. We demonstrated how we used Solidity's `require` statements to enforce security rules in our code, such as verifying auction state, deadlines, user authenticity, and data validity. The implementation comprises a functional prototype of the auction model; hence, this fulfills Research Objective **RO 2**.

In order to evaluate the prototype, we wrote a total of 37 unit tests combined for the smart contracts. These verify that the seller and the bidders of an auction can use the eAuction platform as intended, i.e. that all bidders receive their deposits back at the end of an auction, that the winner can retrieve his token, and that bids are stored on the contract as state variables when submitted. However, the tests also assert that the security rules work as expected, i.e. that bidders that make a fraudulent bid attempt do not receive their deposits back at the end of the auction, that a user that did not win the auction is not allowed to retrieve the winner token, that the respective rounds cannot be

closed before the deadlines expire, and so on. It was also necessary to conduct a security analysis of the solution. This analysis address several of the most known attacks relevant to our implementation, such as replay attacks, denial of service attacks, and man-in-the-middle attacks, with respect to the user’s security requirements. Hence, Research Objective **RO 3** is fulfilled, as the unit tests evaluate the functionality and security of the prototype, and the security analysis assesses possible attacks toward the solution and discusses the risks and consequences for the system’s users.

Our contribution with this thesis provides a deeper insight into how a blind auction can be implemented on a transparent blockchain, benefiting from the security, autonomy, immutability, and decentralization of blockchain technology. This decentralized eAuction platform removes the need for intermediaries such as energy brokers and gives the prosumers and consumers in a microgrid the possibility to gain significant financial benefits; the prosumers by selling their excess renewable energy, and the consumers by obtaining energy from another source than their regional provider, which will significantly lower their monthly energy bills. Furthermore, from our implementation assessment, we conclude that our solution will ensure the eAuction users’ security requirements and that the prototype provides an effective blockchain-based energy trading platform within a microgrid.

6.1 Future Work

At the time when we designed our solution, we considered the construction of the winner’s token out of scope as this relates to the physical aspects of retrieving energy. The token was therefore not carefully designed to be secure within our platform. Since then, we have thought about this problem and come to the conclusion that the token could be generated as a separate smart contract instead of as a state variable in the Auction contract. The token contract would include the winner’s Ethereum account address and the energy amount he won. The contract could then include a function `spendToken()` which would require that the function is called with the winner’s Ethereum account for authenticity, and delete the contract at the end of the function with `selfdestruct`. This would prevent the possibility of double-spending the winner’s token.

Furthermore, the contracts of our solution should be optimized with respect to gas use. Gas price has been considered to a point during the implementation phase of this thesis,

but calculating the optimized bytecode is yet to be executed. This includes utilizing bitwise operations or simple things as structuring the order of variables in a struct. As each struct is tightly packed, this can be calculated on the byte sizes of each variable.

Appendix A

Auction Controller Source Code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.1;
3
4 import "./Auction.sol";
5
6 contract AuctionController {
7     address private admin;
8     mapping(address => address) public sellerAddresses; // ...
9     auction addr => seller addr
10
11     event AddedNewAuction(address auction);
12     event DeletedAuction(address auction);
13
14     constructor() {
15         admin = msg.sender;
16     }
17
18     function deployNewAuction(
19         address payable _seller,
20         uint _energyAmount,
21         uint _minBidValue,
22         uint _depositValue
23     ) public {
24         // Deploy new auction contract
25         Auction newAuction = new Auction(
26             _seller,
```

```

26         _energyAmount,
27         _minBidValue,
28         _depositValue
29     );
30
31     // Save seller address
32     sellerAddresses[address(newAuction)] = _seller;
33     emit AddedNewAuction(address(newAuction));
34 }
35
36 /// Auction cannot be deleted until either:
37 ///     (1) Token has been retrieved
38 ///     (2) Token has expired
39 ///     (3) Auction has closed with no bids
40 /// Auction can only be deleted by admin or by the auction ...
41     seller
42 function deleteAuction(address auctionAddress) public {
43     Auction auction = Auction(auctionAddress);
44
45     require(msg.sender == sellerAddresses[auctionAddress] ...
46         || msg.sender == admin, "Can only be deleted by ...
47         admin or the auction seller");
48
49     bool tokenExpired = block.timestamp > ...
50         auction.getTokenValidUntil() && ...
51         auction.getTokenValidUntil() != 0;
52     if (!tokenExpired) {
53         require(auction.getCurrentState() == ...
54             Auction.State.ReadyForDeletion, "Cannot delete ...
55             auction before the token has expired or been ...
56             retrieved");
57     }
58
59     auction.deleteAuction();
60     delete sellerAddresses[auctionAddress];
61
62     emit DeletedAuction(auctionAddress);
63 }
64 }

```

Appendix B

Auction Source Code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.1;
3
4 contract Auction {
5     enum State {
6         ReadyForHiddenBids,
7         ReadyForOpenBids,
8         Closed,
9         ReadyForDeletion
10    }
11
12    modifier inState(State expectedState) {
13        require(auctionInfo.currentState == expectedState, ...
14               "Invalid state");
15    }
16
17    modifier isBeforeDeadline(uint deadline) {
18        require(block.timestamp < deadline, "Cannot bid after ...
19               deadline");
20    }
21
22    modifier isAfterDeadline(uint deadline) {
23        require(block.timestamp > deadline, "Cannot perform ...
24               this action before the deadline");
```



```

24     -;
25 }
26
27 struct AuctionInfo {
28     State currentState;
29     address payable seller;
30     uint energyAmount;
31     uint minBidValue;
32     uint depositValue;
33     uint hiddenBidsDeadline;
34     uint openBidsDeadline;
35 }
36
37 struct Bid {
38     bool existsHiddenBid;
39     bytes32 hiddenBid;
40     uint openBid;
41     bool isOpenBidValid;
42     uint deposit;
43 }
44
45 struct Winner {
46     address accountAddress;
47     uint bid;
48 }
49
50 struct Token {
51     address winner;
52     address auctionContract;
53     uint energyAmount;
54     uint createdAt;
55     uint validUntil;
56 }
57
58 address private controller;
59 AuctionInfo public auctionInfo;
60 Winner public winner;
61 mapping(address => Bid) public bids;
62 mapping(address => Token) private token;
63 address[] public hiddenBidsAddresses;
64

```

```

65     event CreatedNewAuction(AuctionInfo auctionInfo, uint ...
        currentTime);
66     event ReceivedHiddenBid(address bidder, uint deposit, uint ...
        currentTime);
67     event ReceivedOpenBid(address bidder, uint bid, uint ...
        currentTime);
68     event ClosedRound(string whichRound, State state, uint ...
        currentTime);
69     event ClosedAuctionWithNoBids(string whichRound, uint ...
        currentTime);
70     event FoundHighestBid(Winner winner, uint currentTime);
71     event AuctionEnded(Winner winner, uint contractBalance, ...
        uint currentTime);
72     event TransferEvent(string context, address to, uint value, ...
        uint currentTime);
73     event RetrievedToken(address retrievedBy, uint currentTime);
74
75     // msg.sender is the controller controller and not the ...
        seller address
76     // Seller address must therefore be specified as a parameter
77     constructor(
78         address payable _seller,
79         uint _energyAmount,
80         uint _minBidValue,
81         uint _depositValue
82     ) {
83         uint currentTime = block.timestamp;
84         controller = msg.sender;
85         auctionInfo = AuctionInfo({
86             currentState: State.ReadyForHiddenBids,
87             seller: _seller,
88             energyAmount: _energyAmount,
89             minBidValue: _minBidValue * 1 wei,
90             depositValue: _depositValue * 1 wei,
91             hiddenBidsDeadline: currentTime + 1 days,
92             openBidsDeadline: currentTime + 2 days
93         });
94
95         emit CreatedNewAuction(auctionInfo, currentTime);
96     }
97

```

```

98     /// Place a hidden bid by hashing it with keccak256().
99     /// The deposit is only refunded if the bid is above the ...
    minimum bid value,
100    /// and if the open bid equals the hashed bid during the ...
    open round
101    function bidInHiddenRound(bytes32 bid) public payable
102        inState(State.ReadyForHiddenBids)
103        isBeforeDeadline(auctionInfo.hiddenBidsDeadline)
104    {
105        require(msg.value >= auctionInfo.depositValue, "Deposit ...
            value is too low");
106
107        bids[msg.sender] = Bid({
108            existsHiddenBid: true,
109            hiddenBid: bid,
110            openBid: 0,
111            isOpenBidValid: false,
112            deposit: msg.value * 1 wei
113        });
114
115        hiddenBidsAddresses.push(msg.sender);
116        emit ReceivedHiddenBid(msg.sender, msg.value, ...
            block.timestamp);
117    }
118
119    function closeHiddenRound() public ...
        inState(State.ReadyForHiddenBids) ...
        isAfterDeadline(auctionInfo.hiddenBidsDeadline) {
120        if (hiddenBidsAddresses.length == 0) {
121            auctionInfo.currentState = State.ReadyForDeletion;
122            emit ClosedAuctionWithNoBids("Hidden round", ...
                block.timestamp);
123        } else {
124            auctionInfo.currentState = State.ReadyForOpenBids;
125            emit ClosedRound("Hidden round", ...
                auctionInfo.currentState, block.timestamp);
126        }
127    }
128

```

```

129     function bidInOpenRound(uint openBid, string memory salt) ...
        public inState(State.ReadyForOpenBids) ...
        isBeforeDeadline(auctionInfo.openBidsDeadline) {
130         require(bids[msg.sender].existsHiddenBid, "This account ...
            has not bidden in the hidden round");
131         require(openBid >= auctionInfo.minBidValue, "Bid value ...
            is too low");
132
133         bytes32 hashedBid = keccak256(abi.encodePacked(openBid, ...
            salt));
134         require(bids[msg.sender].hiddenBid == hashedBid, "Open ...
            bid and hidden bid do not match");
135
136         bids[msg.sender].isOpenBidValid = true;
137         bids[msg.sender].openBid = openBid;
138         emit ReceivedOpenBid(msg.sender, openBid, ...
            block.timestamp);
139     }
140
141     function closeAuction() public ...
        isAfterDeadline(auctionInfo.openBidsDeadline) ...
        inState(State.ReadyForOpenBids) {
142         uint validOpenBids = 0;
143         for (uint i = 0; i < hiddenBidsAddresses.length; i++) {
144             if (bids[hiddenBidsAddresses[i]].isOpenBidValid) {
145                 validOpenBids += 1;
146             }
147         }
148
149         if (validOpenBids == 0) {
150             auctionInfo.currentState = State.ReadyForDeletion;
151             emit ClosedAuctionWithNoBids("Open round, no valid ...
                bids", block.timestamp);
152         } else {
153             auctionInfo.currentState = State.Closed;
154             emit ClosedRound("Open round", ...
                auctionInfo.currentState, block.timestamp);
155
156             findWinner();
157         }
158     }

```

```

159
160     function findWinner() internal inState(State.Closed) {
161         address winnerAddress;
162         uint highestBid;
163
164         for(uint i = 0; i < hiddenBidsAddresses.length; i++) {
165             address bidder = hiddenBidsAddresses[i];
166             if (!bids[bidder].isOpenBidValid) continue;
167             uint bid = bids[bidder].openBid;
168
169             if (bid > highestBid) {
170                 winnerAddress = bidder;
171                 highestBid = bid;
172             }
173         }
174
175         winner = Winner({
176             accountAddress: winnerAddress,
177             bid: highestBid
178         });
179         emit FoundHighestBid(winner, block.timestamp);
180
181         token[winnerAddress] = Token({
182             winner: winnerAddress,
183             auctionContract: address(this),
184             energyAmount: auctionInfo.energyAmount,
185             createdAt: block.timestamp,
186             validUntil: block.timestamp + 12 weeks
187         });
188
189         transferBackDeposits();
190     }
191
192     function transferBackDeposits() internal ...
193     inState(State.Closed) {
194         require(winner.accountAddress != address(0), "Must find ...
195             a winner before sending back deposits");
196
197         for (uint i = 0; i < hiddenBidsAddresses.length; i++) {
198             address payable bidderAddress = ...
199                 payable(hiddenBidsAddresses[i]);

```

```

197         Bid memory bid = bids[bidderAddress];
198
199         // Do not send back deposit to invalid bidders
200         if (!bid.isOpenBidValid) continue;
201
202         bool isWinner = bidderAddress == ...
                winner.accountAddress;
203         if (isWinner && bid.openBid >= bid.deposit) continue;
204         uint deposit = isWinner ? bid.deposit - ...
                bid.openBid : bid.deposit;
205
206         emit TransferEvent(
207             "Transfer back deposit to bidder",
208             bidderAddress,
209             deposit,
210             block.timestamp
211         );
212
213         bidderAddress.transfer(deposit);
214     }
215
216     transferHighestBidToSeller();
217 }
218
219 function transferHighestBidToSeller() internal ...
    inState(State.Closed) {
220     uint highestBid = winner.bid;
221     address payable seller = auctionInfo.seller;
222     string memory eventMsg = "Transfer highest bid to seller";
223
224     if (highestBid > auctionInfo.depositValue) {
225         highestBid = auctionInfo.depositValue;
226         eventMsg = "The highest bid was higher than the ...
                deposit value. Transferring the deposit to ...
                seller instead";
227     }
228
229     emit TransferEvent(
230         eventMsg,
231         seller,
232         highestBid,

```

```

233         block.timestamp
234     );
235
236     seller.transfer(highestBid);
237
238     // Transfer deposits of invalid bidders to seller
239     uint contractBalance = address(this).balance;
240     if (contractBalance > 0) {
241         emit TransferEvent(
242             "Transfer contract balance to seller",
243             seller,
244             contractBalance,
245             block.timestamp
246         );
247
248         seller.transfer(contractBalance);
249     }
250
251     emit AuctionEnded(winner, address(this).balance, ...
252         block.timestamp);
253 }
254
255 function retrieveToken() public inState(State.Closed) ...
256     isAfterDeadline(auctionInfo.openBidsDeadline) ...
257     returns(Token memory) {
258     require(msg.sender == winner.accountAddress, "You are ...
259         not the winner of the auction!");
260
261     auctionInfo.currentState = State.ReadyForDeletion;
262     emit RetrievedToken(msg.sender, block.timestamp);
263
264     return token[msg.sender];
265 }
266
267 function getCurrentState() public view returns(State) {
268     return auctionInfo.currentState;
269 }
270
271 function getTokenValidUntil() public view returns(uint) {
272     return token[winner.accountAddress].validUntil;
273 }

```

```
270
271     function deleteAuction() external {
272         require(msg.sender == controller, "You are not allowed ...
           to delete this auction!");
273         selfdestruct(auctionInfo.seller);
274     }
275 }
```


Appendix C

Auction Controller Tests Source Code

```
1 let AuctionController = ...
  artifacts.require("./TestAuctionController.sol");
2 let Auction = artifacts.require("./TestAuction.sol");
3 const truffleAssert = require("truffle-assertions");
4 const { time } = require("@openzeppelin/test-helpers");
5
6 contract("AuctionController", accounts => {
7   let contract;
8   const owner = accounts[0];
9   const seller = accounts[1];
10  const energyAmount = 200;
11  const minBid = 5000000;
12  const deposit = 1000000000;
13  const ONE_DAY = 86400;
14
15  beforeEach(async () => {
16    contract = await AuctionController.new(
17      { from: owner, gas: 6700000 }
18    );
19  });
20
21  it("contract is initialized", async () => {
22    const admin = await contract.testGetAdmin();
```

```

23     expect(admin).to.equal(owner);
24   });
25
26   it("can deploy new auction contract", async () => {
27     const tx = await contract.deployNewAuction(
28       seller,
29       energyAmount,
30       minBid,
31       deposit,
32     );
33
34     const newAuctionAddress = tx.logs[0].args.auction;
35     const sellerAddress = await ...
36       contract.sellerAddresses.call(newAuctionAddress);
37     expect(sellerAddress).to.equal(seller);
38
39     const auction = await Auction.at(newAuctionAddress);
40     const {1: aSeller, 2: aEnergyAmount, 3: aMinBid, 4: ...
41       aDeposit} = await auction.getAuctionInfo();
42     expect(aSeller).to.equal(seller);
43     expect(Number(aEnergyAmount)).to.equal(energyAmount);
44     expect(Number(aMinBid)).to.equal(minBid);
45     expect(Number(aDeposit)).to.equal(deposit);
46
47     truffleAssert.eventEmitted(tx, "AddedNewAuction");
48   });
49
50   it("cannot delete auction if not admin or auction seller", ...
51     async () => {
52       const tx = await contract.deployNewAuction(
53         seller,
54         energyAmount,
55         minBid,
56         deposit,
57       );
58
59       const newAuctionAddress = tx.logs[0].args.auction;
60
61       await truffleAssert.reverts(
62         contract.deleteAuction(newAuctionAddress, { from: ...
63           accounts[2] }),

```

```

60         "Can only be deleted by admin or the auction seller"
61     );
62 });
63
64 it("cannot delete auction if in the hidden round", async () ...
65     => {
66         const tx = await contract.deployNewAuction(
67             seller,
68             energyAmount,
69             minBid,
70             deposit,
71         );
72
73         const newAuctionAddress = tx.logs[0].args.auction;
74
75         await truffleAssert.reverts(
76             contract.deleteAuction(newAuctionAddress, { from: ...
77                 owner }),
78             "Cannot delete auction before the token has expired ...
79                 or been retrieved"
80         );
81     });
82
83 it("cannot delete auction if in the open round", async () ...
84     => {
85         const tx = await contract.deployNewAuction(
86             seller,
87             energyAmount,
88             minBid,
89             deposit,
90         );
91
92         const newAuctionAddress = tx.logs[0].args.auction;
93         const auction = await Auction.at(newAuctionAddress);
94         await ...
95             auction.bidInHiddenRound(web3.utils.soliditySha3(minBid, ...
96                 "some_salt"), { from: accounts[2], value: deposit });
97         await time.increase(ONE_DAY + 1);
98         await auction.closeHiddenRound();
99
100        await truffleAssert.reverts(

```

```

95         contract.deleteAuction(newAuctionAddress, { from: ...
96             owner } ),
97     );
98 });
99
100 it("cannot delete auction if the token has not yet expired ...
101     or been retrieved", async () => {
102     const tx = await contract.deployNewAuction(
103         seller,
104         energyAmount,
105         minBid,
106         deposit,
107     );
108     const newAuctionAddress = tx.logs[0].args.auction;
109     const auction = await Auction.at(newAuctionAddress);
110     await ...
111         auction.bidInHiddenRound(web3.utils.soliditySha3(minBid, ...
112             "some_salt"), { from: accounts[2], value: deposit });
113     await time.increase(ONE_DAY + 1);
114     await auction.closeHiddenRound();
115     await auction.bidInOpenRound(minBid, "some_salt", { ...
116         from: accounts[2] });
117     await time.increase(ONE_DAY + 1);
118     await auction.closeAuction();
119     await truffleAssert.reverts(
120         contract.deleteAuction(newAuctionAddress, { from: ...
121             owner } ),
122         "Cannot delete auction before the token has expired ...
123             or been retrieved"
124     );
125 });
126
127 it("admin can delete auction", async () => {
128     const auctionAddress = await mockAuction(seller);

```

```

126     const deleteTx = await ...
           contract.deleteAuction(auctionAddress, { from: owner ...
           });
127     truffleAssert.eventEmitted(deleteTx, "DeletedAuction", ...
           (ev) => ev.auction == auctionAddress);
128   });
129
130   it("seller can delete auction", async () => {
131     const auctionAddress = await mockAuction(seller);
132
133     const deleteTx = await ...
           contract.deleteAuction(auctionAddress, { from: ...
           seller });
134     truffleAssert.eventEmitted(deleteTx, "DeletedAuction", ...
           (ev) => ev.auction == auctionAddress);
135   });
136
137   it("seller can delete his own auction, but not one from ...
another seller", async () => {
138     const sellersAuction = await mockAuction(seller);
139     const anotherAuction = await mockAuction(accounts[2]);
140
141     const deleteTx = await ...
           contract.deleteAuction(sellersAuction, { from: ...
           seller });
142     truffleAssert.eventEmitted(deleteTx, "DeletedAuction", ...
           (ev) => ev.auction == sellersAuction);
143
144     await truffleAssert.reverts(
145       contract.deleteAuction(anotherAuction, { from: ...
           seller }),
146       "Can only be deleted by admin or the auction seller"
147     );
148   });
149
150   mockAuction = async (sellerAddress) => {
151     const tx = await contract.deployNewAuction(
152       sellerAddress,
153       energyAmount,
154       minBid,
155       deposit,

```

```
156     );
157
158     const newAuctionAddress = tx.logs[0].args.auction;
159     const auction = await Auction.at(newAuctionAddress);
160
161     await ...
162         auction.bidInHiddenRound(web3.utils.soliditySha3(minBid, ...
163             "some_salt"), { from: accounts[2], value: deposit });
164     await time.increase(ONE_DAY + 1);
165     await auction.closeHiddenRound();
166     await auction.bidInOpenRound(minBid, "some_salt", { ...
167         from: accounts[2] });
168     await time.increase(ONE_DAY + 1);
169     await auction.closeAuction();
170     await auction.retrieveToken({ from: accounts[2] });
171     return newAuctionAddress;
172 };
173 });
```

```
> truffle test ./test/auction_controller_test.js
Using network 'development'.

Compiling your contracts...
=====
> Compiling ./contracts/Auction.sol
> Compiling ./contracts/AuctionController.sol
> Compiling ./contracts/TestAuction.sol
> Compiling ./contracts/TestAuctionController.sol
> Artifacts written to /var/folders/mh/446d2mds2zscpkt_1zfbmkl80000gn/T/test--24752-0Qjl1CPYRbEs
> Compiled successfully using:
  - solc: 0.8.1+commit.df193b15.Emscripten.clang

Contract: AuctionController
  ✓ contract is initialized
  ✓ can deploy new auction contract (266ms)
  ✓ cannot delete auction if not admin or auction seller (1120ms)
  ✓ cannot delete auction if in the hidden round (275ms)
  ✓ cannot delete auction if in the open round (431ms)
  ✓ cannot delete auction if the token has not yet expired or been retrieved (606ms)
  ✓ admin can delete auction (692ms)
  ✓ seller can delete auction (570ms)
  ✓ seller can delete his own auction, but not one from another seller (1178ms)

9 passing (6s)
```

Figure C.1: All 9 auction controller tests pass

Appendix D

Auction Tests Source Code

```
1 let Auction = artifacts.require("./TestAuction.sol");
2 const truffleAssert = require("truffle-assertions");
3 const { time } = require("@openzeppelin/test-helpers");
4
5 contract("Auction", accounts => {
6   let contract;
7   const sellerAccount = accounts[9];
8
9   const READY_FOR_HIDDEN_BIDS_STATE = 0;
10  const READY_FOR_OPEN_BIDS_STATE = 1;
11  const CLOSED_STATE = 2;
12  const READY_FOR_DELETION_STATE = 3;
13
14  const ONE_DAY = 86400;
15  const ENERGY_AMOUNT = 200;
16  const MIN_BID_VALUE = 50000;
17  const DEPOSIT_VALUE = 100000;
18
19  const MOCK_BIDS = [MIN_BID_VALUE + 2, MIN_BID_VALUE + 4, ...
20    MIN_BID_VALUE + 1, MIN_BID_VALUE + 3, MIN_BID_VALUE];
21
22  beforeEach(async () => {
23    contract = await Auction.new(
24      sellerAccount,
25      ENERGY_AMOUNT,
26      MIN_BID_VALUE,
```



```

26         DEPOSIT_VALUE,
27         {
28             gas: 4000000
29         }
30     );
31 });
32
33 it("contract is initialized", async () => {
34     let a = await getAuctionInfo();
35     const latestTime = await time.latest();
36
37     expect(a.currentState).to.equal(READY_FOR_HIDDEN_BIDS_STATE);
38     expect(a.seller).to.equal(sellerAccount);
39     expect(a.energyAmount).to.equal(ENERGY_AMOUNT);
40     expect(a.minBidValue).to.equal(MIN_BID_VALUE);
41     expect(a.depositValue).to.equal(DEPOSIT_VALUE);
42     expect(a.hiddenBidsDeadline).to.equal(latestTime.toNumber() ...
43         + ONE_DAY);
44     expect(a.openBidsDeadline).to.equal(latestTime.toNumber() ...
45         + (ONE_DAY * 2));
46 });
47
48 // TESTS DURING HIDDEN ROUND
49
50 it("can bid in hidden round", async () => {
51     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
52         DEPOSIT_VALUE);
53     const bid = await contract.bids.call(accounts[1]);
54
55     expect(bid.existsHiddenBid).to.equal(true);
56     expect(Number(bid.deposit)).to.equal(DEPOSIT_VALUE);
57 });
58
59 it("cannot bid in hidden round if in the wrong state", ...
60     async () => {
61         await contract.setCurrentState(READY_FOR_OPEN_BIDS_STATE);
62         await truffleAssert.reverts(
63             bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
64                 DEPOSIT_VALUE),
65             "Invalid state"
66         );

```

```

62     });
63
64     it("cannot bid in hidden round if after deadline", async () ...
        => {
65         await time.increase(ONE_DAY + 1);
66         await truffleAssert.reverts(
67             bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
                DEPOSIT_VALUE),
68             "Cannot bid after deadline"
69         );
70     });
71
72     it("cannot bid in hidden round if deposit is too low", ...
        async () => {
73         await truffleAssert.reverts(
74             bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
                DEPOSIT_VALUE - 1),
75             "Deposit value is too low"
76         );
77     });
78
79     it("auction should close if no hidden bids were received", ...
        async () => {
80         await time.increase(ONE_DAY + 1);
81
82         const tx = await contract.closeHiddenRound();
83         truffleAssert.eventEmitted(tx, ...
            "ClosedAuctionWithNoBids", (ev) => {
84             return ev.whichRound == "Hidden round";
85         });
86
87         const hiddenBidsLength = Number(await ...
            contract.getHiddenBidsLength());
88         expect(hiddenBidsLength).to.equal(0);
89
90         const state = Number(await contract.getCurrentState());
91         expect(state).to.equal(READY_FOR_DELETION_STATE);
92     });
93
94     it("closed hidden round and started open round", async () ...
        => {

```

```

95     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
          DEPOSIT_VALUE);
96     await time.increase(ONE_DAY + 1);
97     const tx = await contract.closeHiddenRound();
98
99     truffleAssert.eventEmitted(tx, "ClosedRound", (ev) => ...
          ev.whichRound == "Hidden round");
100
101     let a = await getAuctionInfo();
102     expect(a.currentState).to.equal(READY_FOR_OPEN_BIDS_STATE);
103   });
104
105   it("cannot close hidden round if in the wrong state", async ...
        () => {
106     await contract.setCurrentState(READY_FOR_OPEN_BIDS_STATE);
107     await truffleAssert.reverts(
108       contract.closeHiddenRound(),
109       "Invalid state"
110     );
111   });
112
113   it("cannot close hidden round before deadline", async () => {
114     await time.increase(ONE_DAY - 1);
115     await truffleAssert.reverts(
116       contract.closeHiddenRound(),
117       "Cannot perform this action before the deadline"
118     );
119   });
120
121   // TESTS DURING OPEN ROUND
122
123   it("can bid in open round", async () => {
124     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
          DEPOSIT_VALUE);
125     await time.increase(ONE_DAY + 1);
126     await contract.closeHiddenRound();
127     await contract.bidInOpenRound(MIN_BID_VALUE, ...
          "some_salt", { from: accounts[1] });
128
129     let bid = await contract.bids.call(accounts[1]);
130

```

```

131     expect (bid.isOpenBidValid).to.equal(true);
132     expect (Number(bid.openBid)).to.equal(MIN_BID_VALUE);
133   });
134
135   it("cannot bid in open round if in the wrong state", async ...
      () => {
136     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
        DEPOSIT_VALUE);
137     await time.increase(ONE_DAY + 1);
138
139     await truffleAssert.reverts(
140       contract.bidInOpenRound(MIN_BID_VALUE, { from: ...
        accounts[1] }),
141       "Invalid state"
142     );
143   });
144
145   it("cannot bid in open round if after deadline", async () ...
      => {
146     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
        DEPOSIT_VALUE);
147     await time.increase(ONE_DAY + 1);
148     await contract.closeHiddenRound();
149     await time.increase(ONE_DAY + 1);
150
151     await truffleAssert.reverts(
152       contract.bidInOpenRound(MIN_BID_VALUE, { from: ...
        accounts[1] }),
153       "Cannot bid after deadline"
154     );
155   });
156
157   it("cannot bid in open round if not already bidden in ...
      hidden round", async () => {
158     await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
        DEPOSIT_VALUE);
159     await time.increase(ONE_DAY + 1);
160     await contract.closeHiddenRound();
161
162     await truffleAssert.reverts(

```

```

163         contract.bidInOpenRound(MIN_BID_VALUE, "some_salt", ...
164             { from: accounts[2] } ),
165         "This account has not bidden in the hidden round"
166     );
167 });
168 it("cannot bid in open round if bid is too low", async () ...
169     => {
170         await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
171             DEPOSIT_VALUE);
172         await time.increase(ONE_DAY + 1);
173         await contract.closeHiddenRound();
174
175         await truffleAssert.reverts(
176             contract.bidInOpenRound(MIN_BID_VALUE - 1, ...
177                 "some_salt", { from: accounts[1] } ),
178             "Bid value is too low"
179         );
180     });
181 it("cannot bid in open round if bid does not match hidden ...
182     bid", async () => {
183         await bidInHiddenRound(MIN_BID_VALUE, accounts[1], ...
184             DEPOSIT_VALUE);
185         await time.increase(ONE_DAY + 1);
186         await contract.closeHiddenRound();
187
188         await truffleAssert.reverts(
189             contract.bidInOpenRound(MIN_BID_VALUE + 1, ...
190                 "some_salt", { from: accounts[1] } ),
191             "Open bid and hidden bid do not match"
192         );
193     });
194 it("auction should close if no valid open bids were ...
195     recevied", async () => {
196         await bidInHiddenRound(MIN_BID_VALUE + 1, accounts[1], ...
197             DEPOSIT_VALUE);
198         await time.increase(ONE_DAY + 1);
199         await contract.closeHiddenRound();

```

```

195     await truffleAssert.reverts(
196         contract.bidInOpenRound(MIN_BID_VALUE, "some_salt", ...
197             { from: accounts[1]}),
198         "Open bid and hidden bid do not match"
199     );
200
201     await time.increase(ONE_DAY + 1);
202     const tx = await contract.closeOpenRound();
203     truffleAssert.eventEmitted(tx, ...
204         "ClosedAuctionWithNoBids", (ev) => {
205         return ev.whichRound == "Open round, no valid bids";
206     });
207
208     const state = Number(await contract.getCurrentState());
209     expect(state).to.equal(READY_FOR_DELETION_STATE);
210
211     truffleAssert.eventEmitted(tx, ...
212         "ClosedAuctionWithNoBids", (ev) => ev.whichRound == ...
213         "Open round, no valid bids");
214
215     });
216
217     it("closed open round", async () => {
218         await mockBidding(MOCK_BIDS);
219         await time.increase(ONE_DAY + 1);
220         // await ...
221         contract.setCurrentState(READY_FOR_OPEN_BIDS_STATE);
222         const tx = await contract.closeOpenRound();
223
224         let a = await getAuctionInfo();
225         expect(a.currentState).to.equal(CLOSED_STATE);
226         truffleAssert.eventEmitted(tx, "ClosedRound", (ev) => ...
227             ev.whichRound == "Open round");
228     });
229
230     it("cannot close open round if in the wrong state", async ...
231         () => {
232         await time.increase((ONE_DAY * 2) + 1);
233         await truffleAssert.reverts(
234             contract.closeOpenRound(),
235             "Invalid state"
236         )

```

```

229     });
230
231     it("cannot close open round before deadline", async () => {
232         await contract.setCurrentState(READY_FOR_OPEN_BIDS_STATE);
233         await truffleAssert.reverts(
234             contract.closeOpenRound(),
235             "Cannot perform this action before the deadline"
236         )
237     });
238
239     // TESTS FOR CLOSING THE AUCTION
240
241     it("found auction winner", async () => {
242         const actualHighestBid = await mockBidding(MOCK_BIDS);
243         await time.increase(ONE_DAY + 1);
244         await contract.closeOpenRound();
245         const tx = await contract.testFindWinner();
246
247         const winner = await contract.winner.call();
248
249         truffleAssert.eventEmitted(tx, "FoundHighestBid");
250         expect(winner.accountAddress).to.equal(actualHighestBid.bidder);
251         expect(Number(winner.bid)).to.equal(actualHighestBid.bid);
252     });
253
254     it("cannot find auction winner if in wrong state", async () ...
255         => {
256         await mockBidding(MOCK_BIDS);
257         await time.increase(ONE_DAY + 1);
258
259         await truffleAssert.reverts(
260             contract.testFindWinner(),
261             "Invalid state"
262         );
263     });
264
265     it("sent deposits back to bidders (all bids valid)", async ...
266         () => {
267         const highestBid = await mockBidding(MOCK_BIDS);
268         await time.increase(ONE_DAY + 1);
269         await contract.closeOpenRound();

```

```

268     await contract.testFindWinner();
269
270     const winner = await contract.winner.call();
271     expect(winner.accountAddress).to.equal(highestBid.bidder);
272     expect(Number(winner.bid)).to.equal(highestBid.bid);
273
274     let balancesBefore = [];
275     for (let i = 0; i < MOCK_BIDS.length; i++) {
276         balancesBefore.push(await getBalance(accounts[i + ...
277             1]));
278     }
279
280     const tx = await contract.testTransferBackDeposits();
281     truffleAssert.eventEmitted(tx, "TransferEvent");
282
283     for (let i = 0; i < MOCK_BIDS.length; i++) {
284         const isWinner = accounts[i + 1] === ...
285             winner.accountAddress;
286         const currentBalance = await getBalance(accounts[i ...
287             + 1]);
288
289         const refundedValue = isWinner ? DEPOSIT_VALUE - ...
290             MOCK_BIDS[i] : DEPOSIT_VALUE;
291         expect(Number(currentBalance - ...
292             balancesBefore[i])).to.equal(refundedValue);
293     }
294 }
295 });
296
297 it("did not send deposit back to invalid bidder", async () ...
298 => {
299     const invalidBidder = accounts[1];
300     await mockBidding(MOCK_BIDS, true); // Include invalid ...
301         first bid
302
303     await contract.setCurrentState(CLOSED_STATE);
304     await contract.testFindWinner();
305
306     let balanceBefore = await getBalance(invalidBidder);
307     let tx = await contract.testTransferBackDeposits();
308     truffleAssert.eventEmitted(tx, "TransferEvent");
309     let balanceAfter = await getBalance(invalidBidder);

```



```

302     expect(Number(balanceAfter - balanceBefore)).to.equal(0);
303   });
304
305   it("sent highest bid to seller, no extra deposits", async ...
      () => {
306     const highestBid = await mockBidding(MOCK_BIDS);
307     await time.increase(ONE_DAY + 1);
308     await contract.closeOpenRound();
309     await contract.testFindWinner();
310     await contract.testTransferBackDeposits();
311
312     const balanceBefore = await getBalance(sellerAccount);
313     const tx = await ...
        contract.testTransferHighestBidToSeller();
314     truffleAssert.eventEmitted(tx, "TransferEvent");
315     const balanceAfter = await getBalance(sellerAccount);
316
317     expect(Number(balanceAfter - ...
        balanceBefore)).to.equal(highestBid.bid);
318   });
319
320   it("sent highest bid to seller, one extra deposit", async ...
      () => {
321     const highestBid = await mockBidding(MOCK_BIDS, true); ...
        // Include invalid first bid
322     await time.increase(ONE_DAY + 1);
323     await contract.closeOpenRound();
324     await contract.testFindWinner();
325     await contract.testTransferBackDeposits();
326
327     const balanceBefore = BigInt(await ...
        web3.eth.getBalance(sellerAccount));
328     const tx = await ...
        contract.testTransferHighestBidToSeller({ gasPrice: ...
        0});
329     truffleAssert.eventEmitted(tx, "TransferEvent");
330     const balanceAfter = BigInt(await ...
        web3.eth.getBalance(sellerAccount));
331

```

```

332     expect(Number(balanceAfter - ...
333         balanceBefore)).to.equal(highestBid.bid + ...
334         DEPOSIT_VALUE);
335 });
336
337 it("winner retrieved token", async() => {
338     await mockBidding(MOCK_BIDS);
339     await time.increase(ONE_DAY + 1);
340     await contract.closeAuction();
341     const winner = await contract.winner.call();
342
343     const tx = await contract.retrieveToken({ from: ...
344         winner.accountAddress });
345     truffleAssert.eventEmitted(tx, "RetrievedToken");
346 });
347
348 it("non-winner is not allowed to retrieve token", async() ...
349     => {
350         await mockBidding(MOCK_BIDS);
351         await time.increase(ONE_DAY + 1);
352         await contract.closeAuction();
353
354         await truffleAssert.reverts(
355             contract.retrieveToken({ from: accounts[1] }),
356             "You are not the winner of the auction!");
357     });
358
359 it("token should not be callable", async() => {
360     await mockBidding(MOCK_BIDS);
361     await time.increase(ONE_DAY + 1);
362     await contract.closeAuction();
363
364     try {
365         await contract.token.call();
366         expect.fail();
367     } catch(error) {
368         expect(error.message).to.equal("Cannot read ...
369             property 'call' of undefined");
370     }
371 });

```

```

368
369 // CONVENIENCE FUNCTIONS
370
371 getAuctionInfo = async () => {
372     let info = await contract.getAuctionInfo.call();
373     return {
374         "currentState": Number(info[0]),
375         "seller": info[1],
376         "energyAmount": Number(info[2]),
377         "minBidValue": Number(info[3]),
378         "depositValue": Number(info[4]),
379         "hiddenBidsDeadline": Number(info[5]),
380         "openBidsDeadline": Number(info[6]),
381     };
382 }
383
384 bidInHiddenRound = async (bidValue, bidderAddress, ...
depositValue) => {
385     let tx = await ...
contract.bidInHiddenRound(web3.utils.soliditySha3(bidValue, ...
"some_salt"), {
386         value: depositValue,
387         from: bidderAddress
388     });
389
390     truffleAssert.eventEmitted(tx, "ReceivedHiddenBid", ...
(ev) => {
391         return ev.bidder == bidderAddress && ev.deposit == ...
depositValue;
392     });
393 };
394
395 bidInOpenRound = async (bidValue, salt, bidderAddress) => {
396     let tx = await contract.bidInOpenRound(bidValue, salt, {
397         from: bidderAddress
398     });
399
400     truffleAssert.eventEmitted(tx, "ReceivedOpenBid", (ev) ...
=> {
401         return ev.bidder == bidderAddress && ev.bid == ...
bidValue;

```

```

402     });
403   };
404
405   mockBidding = async (bids, includeInvalidBid = false) => {
406     for (let i = 0; i < bids.length; i++) {
407       await bidInHiddenRound(bids[i], accounts[i + 1], ...
         DEPOSIT_VALUE);
408     }
409     let hiddenBidsNum = Number(await ...
       contract.getHiddenBidsLength.call());
410     expect(hiddenBidsNum).to.equal(bids.length);
411
412     await time.increase(ONE_DAY + 1);
413     await contract.closeHiddenRound();
414
415     for (let i = 0; i < bids.length; i++) {
416       if (includeInvalidBid && i == 0) {
417         await truffleAssert.reverts(
418           bidInOpenRound(bids[i] - 1, "some_salt", ...
             accounts[i + 1]),
419           "Open bid and hidden bid do not match"
420         );
421         continue;
422       }
423       await bidInOpenRound(bids[i], "some_salt", ...
         accounts[i + 1]);
424     }
425
426     let highestBid = Math.max(...bids);
427     let highestBidder = accounts[bids.indexOf(highestBid) + ...
       1];
428     return { "bid": highestBid, "bidder": highestBidder};
429   };
430
431   getBalance = async (account) => {
432     return BigInt(await web3.eth.getBalance(account));
433   };
434 });

```

```

> truffle test ./test/auction_test.js
Using network 'development'.

Compiling your contracts...
=====
> Compiling ./contracts/Auction.sol
> Compiling ./contracts/AuctionController.sol
> Compiling ./contracts/TestAuction.sol
> Compiling ./contracts/TestAuctionController.sol
> Artifacts written to /var/folders/mh/446d2mds2zscpkt_1zfbmkl80000gn/T/test--64253-ZdVPnVoXlRYn
> Compiled successfully using:
  - solc: 0.8.1+commit.df193b15.Emscripten.clang

Contract: Auction
  ✓ contract is initialized (60ms)
  ✓ can bid in hidden round (142ms)
  ✓ cannot bid in hidden round if in the wrong state (1458ms)
  ✓ cannot bid in hidden round if after deadline (142ms)
  ✓ cannot bid in hidden round if deposit is too low (72ms)
  ✓ auction should close if no hidden bids were received (354ms)
  ✓ closed hidden round and started open round (371ms)
  ✓ cannot close hidden round if in the wrong state (261ms)
  ✓ cannot close hidden round before deadline (133ms)
  ✓ can bid in open round (614ms)
  ✓ cannot bid in open round if in the wrong state (178ms)
  ✓ cannot bid in open round if after deadline (471ms)
  ✓ cannot bid in open round if not already bidden in hidden round (354ms)
  ✓ cannot bid in open round if bid is too low (346ms)
  ✓ cannot bid in open round if bid does not match hidden bid (438ms)
  ✓ auction should close if no valid open bids were received (603ms)
  ✓ closed open round (1559ms)
  ✓ cannot close open round if in the wrong state (166ms)
  ✓ cannot close open round before deadline (175ms)
  ✓ found auction winner (2106ms)
  ✓ cannot find auction winner if in wrong state (1946ms)
  ✓ sent deposits back to bidders (all bids valid) (3282ms)
  ✓ did not send deposit back to invalid bidder (2501ms)
  ✓ sent highest bid to seller, no extra deposits (3278ms)
  ✓ sent highest bid to seller, one extra deposit (3129ms)
  ✓ winner retrieved token (2228ms)
  ✓ non-winner is not allowed to retrieve token (2255ms)
  ✓ token should not be callable (1866ms)

28 passing (37s)

```

Figure D.1: All 28 auction tests pass

Bibliography

- [1] Charles Kirubi et al. “Community-Based Electric Micro-Grids Can Contribute to Rural Development: Evidence from Kenya.” In: *World Development* 37.7 (2009), pp. 1208–1221. ISSN: 0305-750X. DOI: <https://doi.org/10.1016/j.worlddev.2008.11.005>. URL: <https://www.sciencedirect.com/science/article/pii/S0305750X08003288>.
- [2] Joel Spaes. “Harmon’Yeu, the first energy community on Île d’Yeu, signed by Engie.” In: (2020). URL: <https://www.pv-magazine.fr/2020/07/03/harmonyeu-premiere-communaute-energetique-a-lile-dyeu-signe-engie/>.
- [3] Maxim Buevich et al. “Fine-grained remote monitoring, control and pre-paid electrical service in rural microgrids.” In: (2014), pp. 1–11. DOI: [10.1109/IPSN.2014.6846736](https://doi.org/10.1109/IPSN.2014.6846736).
- [4] Alireza Aram. “Microgrid Market in the USA.” In: (2017). URL: https://www.hitachi.com/rev/archive/2017/r2017_05/Global/index.html.
- [5] Dan T. Ton and Merrill A. Smith. “The U.S. Department of Energy’s Microgrid Initiative.” In: (2012). DOI: [10.1016/j.tej.2012.09.013](https://doi.org/10.1016/j.tej.2012.09.013).
- [6] JOHN G. KASSAKIAN et al. “The Future of the Electricity Grid.” In: (2011).
- [7] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System.” In: *Cryptography Mailing list at https://metzdowd.com* (2008).
- [8] Martin Onyeka Okoye et al. “A Blockchain-Enhanced Transaction Model for Microgrid Energy Trading.” In: (2019).
- [9] IEA-ETSAP IRENA. “Renewable Energy Integration in Power Grids.” In: *IRENA* (2015).

- [10] Erik-Oliver Blass and Florian Kerschbaum. “Strain: A Secure Auction for Blockchains.” In: *IACR Cryptol. ePrint Arch.* (2017).
- [11] S. Chen Y. Chen and I. Lin. “Blockchain based smart contract for bidding system.” In: (2018).
- [12] Maha Kadadha et al. “ABCrowd: An Auction Mechanism on Blockchain for Spatial Crowdsourcing.” In: (2020).
- [13] B. P. Hayes S. Thakur and J. G. Breslin. “Distributed Double Auction for Peer to Peer Energy Trade using Blockchains.” In: (2018).
- [14] Chaum D. “Blind Signatures for Untraceable Payments.” In: *Advances in Cryptology Proceedings of Crypto* (1982).
- [15] Wei Dai. “b-money.” In: (1998).
- [16] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System.” In: *Cryptography Mailing list at <https://metzdowd.com>* (2008).
- [17] Ethereum foundation. *Ethereum Whitepaper*.
- [18] Ethereum. *Ethereum accounts*. URL: <https://ethereum.org/en/developers/docs/accounts/>.
- [19] GAVIN WOOD. “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER.” In: *ETHEREUM & PARITY* (2021).
- [20] Nick Szabo. “Smart Contracts.” In: (1994).
- [21] Nick Szabo. “The Idea of Smart Contracts.” In: (1997).
- [22] Yang Lu. “The blockchain: State-of-the-art and research challenges.” In: (2019).
- [23] Maher Alharby and Aad van Moorsel. “Blockchain-based smart contracts: a systematic mapping study.” In: (2017), pp. 125–140. DOI: [10.5121/cs.it.2017.71011](https://doi.org/10.5121/cs.it.2017.71011).
- [24] Shafi Goldwasser and Mihir Bellare. “Lecture Notes on Cryptography.” In: (2008).
- [25] Solidity. *Solidity*. URL: <https://docs.soliditylang.org/en/v0.8.4/>.
- [26] Ting Chen et al. “Under-optimized smart contracts devour your money.” In: (2017), pp. 442–446. DOI: [10.1109/SANER.2017.7884650](https://doi.org/10.1109/SANER.2017.7884650).
- [27] Solidity. *Mapping Types*. URL: <https://docs.soliditylang.org/en/v0.8.4/types.html#mapping-types>.

- [28] Solidity. *Units and Globally Available Variables*. URL: <https://docs.soliditylang.org/en/latest/units-and-global-variables.html?highlight=block#block-and-transaction-properties>.
- [29] Web3.js. *web3.js - Ethereum JavaScript API*. URL: <https://web3js.readthedocs.io/en/v1.2.6/>.
- [30] MetaMask. *MetaMask - A crypto wallet & gateway to blockchain apps*. URL: <https://metamask.io>.
- [31] Creately. *What is Sequence Diagram? Complete Guide with Examples*. URL: <https://creately.com/blog/diagrams/sequence-diagram-tutorial/>.
- [32] Solidity. *Layout of State Variables in Storage*. URL: https://docs.soliditylang.org/en/v0.8.4/internals/layout_in_storage.html.
- [33] Rosco Kalis. *Truffle Assertions*. URL: <https://www.npmjs.com/package/truffle-assertions>.
- [34] Rosco Kalis. *OpenZeppelin Test Helpers*. URL: <https://github.com/OpenZeppelin/openzeppelin-test-helpers>.
- [35] Huashan Chen et al. "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses." In: (2019). eprint: 1908.04507.