Property Properly: A Decentralized Application for Property Transaction Management

by

Ashlynn Kitatake-Meyers

A Dissertation

Presented to the University of Dublin, Trinity College

in fulfilment of the requirements for the Degree of

Master of Science in Computer Science (Data Science)

University of Dublin, Trinity College

Supervisor: Donal O’Mahony

August 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

____

Ashlynn Kitatake-Meyers

September 14, 2021

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ashlynn Kitatake-Meyers

September 14, 2021

Property Properly: A Decentralized Application for Property Transaction Management

Ashlynn Kitatake-Meyers, Master of Science in Computer Science
University of Dublin, Trinity College, 2021

Supervisor: Donal O'Mahony

The Ethereum blockchain is a platform where programmers can create their own decentralized applications (DApps) using smart contracts. One popular area for modern blockchain development is in exploring applications to the real estate market. Several start-up companies have created DApps specific to property transactions. There are two key advantages of the blockchain that make it uniquely applicable to creating a property transaction management DApp. First, the *immutability* of transactions once they are recorded on the blockchain makes it the perfect technology for property and tenant registries. Second, the automated nature of smart contracts creates a *trustless* environment for various property payments.

In this dissertation, we combine the functionalities of a property registry, tenant registry, and payment system into one DApp. This is done by creating interconnected smart contracts for a property registry, tenant registry, and properties that allow users to initiate transactions and send and receive payments. This project seeks to demonstrate the effectiveness and potential of blockchain development in the real estate market. Future work could seek to deepen the complexity and breadth of the property transactions handled.

Acknowledgements

I would like to sincerely thank my dissertation supervisor Professor Donal O'Mahony for his expertise and guidance throughout this project. His questions and advice were incredibly thoughtful and helped to steer my project focus.

I would also like to thank my mother and my partner for their continued support and encouragement through this degree and research.

ASHLYNN KITATAKE-MEYERS

University of Dublin, Trinity College
August 2021

List of Figures

Figure 1: Block Structure of the Blockchain 5

Figure 2: Merkle Tree Hashing 6

Figure 3: Smart Contract Lifecycle 14

Figure 4: Gas Use From Transactions 18

Figure 5: Basic Structure of the DApp Diagram 33

Figure 6: DApp Back End Diagram 37

Figure 7: DApp User Interface 55

Contents

Abstract iii

Acknowledgements iv

List of Figures v

Chapter 1 Introduction 1
- 1.1 Research Question ... 1
- 1.2 Dissertation Layout ... 2

Chapter 2 State of the Art 3
- 2.1 Blockchain ... 3
  - 2.1.1 Basic Structure ... 3
  - 2.1.2 History ... 8
  - 2.1.3 Cryptocurrency ... 10
  - 2.1.4 Smart Contracts ... 12
  - 2.1.5 Decentralized Applications (DApps) ... 16
  - 2.1.6 Ethereum ... 17
  - 2.1.7 ERC-20 Tokens ... 19
  - 2.1.8 Non-fungible Tokens (NFTs) ... 21
- 2.2 Real Estate ... 24
  - 2.2.1 Basic Structure in the USA ... 24
  - 2.2.2 Industry Participants ... 27
  - 2.2.3 Areas Open to Blockchain ... 29
  - 2.2.4 Current Companies ... 30

Chapter 3 Design 33
- 3.1 Summary of the Approach ... 33

Chapter 4 Implementation 37
- 4.1 Overview of the DApp ... 37
- 4.2 Smart Contracts ... 37
  - 4.2.1 Property Registry ... 38
  - 4.2.2 Property ... 41
  - 4.2.3 Tenant Registry ... 44
- 4.3 User Interface ... 48

Chapter 5 Evaluation 57
- 5.1 Security Considerations ... 57
- 5.2 Solidity Testing ... 58

5.3 Strengths ……………………………………………………………………..59
5.4 Limitations ……………………………………………………………………..60

Chapter 6 Conclusions & Future Work **62**
6.1 Conclusion ……………………………………………………………………...62
6.2 Future Work ……………………………………………………………………63

Bibliography **65**

Chapter 1

Introduction

At its core, the decentralized, immutable, trustless nature of blockchain technology has enormous potential. One such blockchain network, Ethereum, was made specifically to foster customized development and automation of tasks on the blockchain (Buterin 2013). The focus of this dissertation is on developing an application on the Ethereum blockchain to tackle the problem of outdated property ledger and transaction management systems. The key advantages of blockchain make it uniquely applicable to this task. This introductory chapter is split into two sections. The first section will lay out the research question and the second section will give an overview of the structure of this document.

1.1 Research Question

The central problem that this project seeks to solve is that *most property registries are unreliable and inaccessible* -- blockchain has the potential to solve this. In order to explore and generate a solution to this problem, we have created the following research question:

How do we create an automated property ledger and transaction management system on the blockchain?

This larger question also involves many sub-questions:

1. Which blockchain is best for developing this specific application?
2. Which data should we store in the blockchain and which should be stored locally?
3. How do we allow owners to make payments on their properties to other parties? How do we allow tenants to make rent payments to property owners?
4. Which components of the system should have their own smart contracts and how should they be structured?
5. How should we store metadata and other information about the properties?
6. Have we effectively taken advantage of the unique features of the blockchain to accomplish our goal?

1.2 Dissertation Layout

Chapter 1 of this dissertation provides a general introduction of the topic and the specific research question of the project.

Chapter 2 is the state of the art which contains background information on the topic, technologies being used, and current research or developments in the area.

Chapter 3 gives an overview of the design of the project.

Chapter 4 describes the implementation of the project in detail including the functionality of each component. This chapter also contains information on the challenges faced throughout the project implementation and how they were overcome.

Chapter 5 explains the testing that was used as well as the strengths and limitations of the project.

Chapter 6 states the conclusions of the dissertation and potential areas for future work.

Chapter 2

State of the Art

This chapter is split into two sections. The first section will focus on the structure and history of blockchain. It will also go into more detail on the methods by which blockchain development is done, including topics such as the Ethereum blockchain, smart contracts, and decentralized applications. The second section will provide some background on the inner workings of the real estate market and potential areas open to blockchain technology. We will also discuss current companies that are developing systems to fill these unique niches.

2.1 Blockchain

2.1.1 Basic Structure

At its core, blockchain is a digital distributed ledger of transactions (Zheng et al. 2017). A regular ledger is a registry of transactions between participants that are typically recorded by a third party. These registries can range from public to completely private. For example, a patient's health care data might be stored on a private ledger by a hospital. However, marriage licenses or records of death are stored on public ledgers by the government. Another important use of ledgers is in recording property transactions to be able to know who owns a property, who were the previous owners, when was it last sold and more.

Today, most companies and organizations keep digital ledgers. These differ from a physical ledger in that they are recorded digitally. This could be on a private computer, in a company data center, or on the blockchain. However, the blockchain is not only a digital ledger, but also a *distributed* ledger (Zheng et al. 2018).

Being a distributed ledger means that all transaction records are shared across a network of different computers. Every computer on the network has a complete copy of the

ledger and receives all new entries to keep the ledger up to date. In order for a change to be added to the ledger, it needs to be verified by the majority of the computers on the network. This means that every member of the network is able to see exactly what transactions are being added. These transactions are recorded in chronological order and once a transaction is added to the blockchain, it cannot be removed. The security of this is verified by cryptographic signatures and the distributed nature of the blockchain (Zheng et al. 2018). To clarify, if a malicious actor were to fraudulently manipulate the transactions in their copy of the blockchain, it would never be accepted by the other members of the network. This transparency and immutability of the blockchain is a major strength (Zheng et al. 2017).

The method by which this entry verification happens is by a cryptographic algorithm and hash function, which we will discuss below. The key takeaway is that the blockchain is a very secure and efficient implementation of a digital distributed ledger. Every computer in the network is connected via the internet and uses this to verify and add new transactions. There is no owner of the ledger and due to its distributed structure, it is called *decentralized* (Zheng et al. 2018).

For those who have seen the 2018 musical film *Mary Poppins Returns*, the value of a decentralized ledger is abundantly clear. In the film, the father character does indeed own enough bank shares to save the family's home from foreclosure, but this is only recorded in the bank's private ledger. The *centralized* nature of this ledger forces our protagonist to be woefully dependent upon the goodwill of the bank president to honor the private ledger, and is unable to independently verify his ownership of the bank's shares. As shown in the film, this opens the door for a malicious actor, such as the bank president whose motivations are skewed, to refuse to verify the ownership of the shares without any repercussions (Marshall). If the bank ledger had been recorded on the blockchain, the ownership of those bank shares would be public and verified knowledge and the crisis would have been avoided.

It is important to remember that the ledger is not limited to financial transactions (e.g., selling Bitcoin). Theoretically, any transaction could be recorded on the blockchain. Many non-financial transactions have been recorded with varying levels of success thus far. Some examples include votes, medical data, securities, and various cryptocurrencies (Marr) (Domingo). All of these examples are of digital or intangible data, but many have also

attempted to use the blockchain to record transactions of *physical* assets. The two most prominent examples of this are art pieces and properties. The blockchain can be used to record who owns the assets, a transfer of ownership, etc (Karayaneva) (Suum Cuique Labs GmbH) (Smith).

All of these transactions will be available to every member computer (or node) on the network as they all contain a full copy of the blockchain. However, it is also possible for anyone to view transactions without being part of the network - they simply can't add to it. With the Ethereum blockchain, anyone with internet connections can search for transactions on the Ethereum blockchain using etherscan.io (Etherscan).

In addition, the way in which these transactions are carried out guarantees what is called a *trustless* environment. Smart contracts make it so that the transfer of digital assets between parties is done automatically when previously agreed to conditions are met by both parties. Smart contracts will be discussed in more detail below. The key takeaway is that neither party needs to trust the other, they simply must trust the code of the smart contract (Zheng et al. 2019).

With all of this background understood, we will move on to how the blockchain is in fact a *chain of blocks*. The structure of a blockchain varies slightly between implementations, but all follow the same general structure.

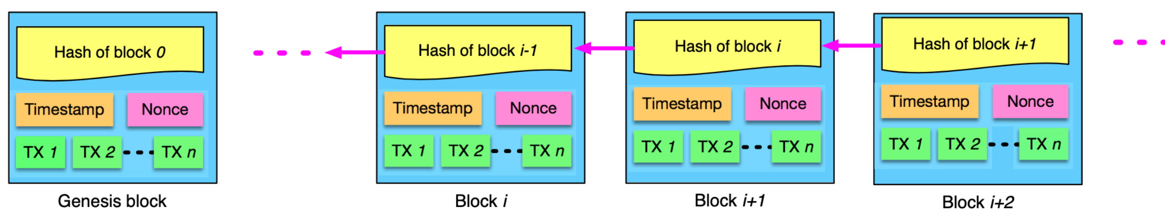

Figure 1: Block Structure of the Blockchain (Zheng et al. 2018)

The blockchain begins with a *genesis block* which is the origin block of the entire chain - it has no predecessor. Every block is connected only to its parent block via a cryptographic algorithm in which the input is the parent block and the output is the child block. The one way nature of this is part of what makes the blockchain so secure in that

knowing the output does not allow you to know the input. The blocks are added in chronological order, and every block may contain one or more transactions (Zheng et al. 2018).

Each block contains a transaction counter (the number of transactions that block can record) and the content of the transactions themselves. In addition, the block contains a version number which specifies how the block should be validated. Next, the block contains a 256-bit hash value that is it's connection to the parent block. The block also contains the transaction root hash which will be discussed in more detail below. Then there is a timestamp and a nonce value, which represents the proof of work done. This concept will also be explained below (Zheng et al. 2018).

The transaction root hash is created using a Merkle Tree where the root is the root hash specified. The ends of the merkle tree are the hashes of the individual transactions. These transactions are then paired up and hashed together. This process is repeated until there is one final hash - the root hash. This is what creates the tree structure. This structure also allows the transaction data to be carried across the blockchain efficiently and with little storage requirements (Zheng et al. 2018). This is shown in the figure below (Gupta).

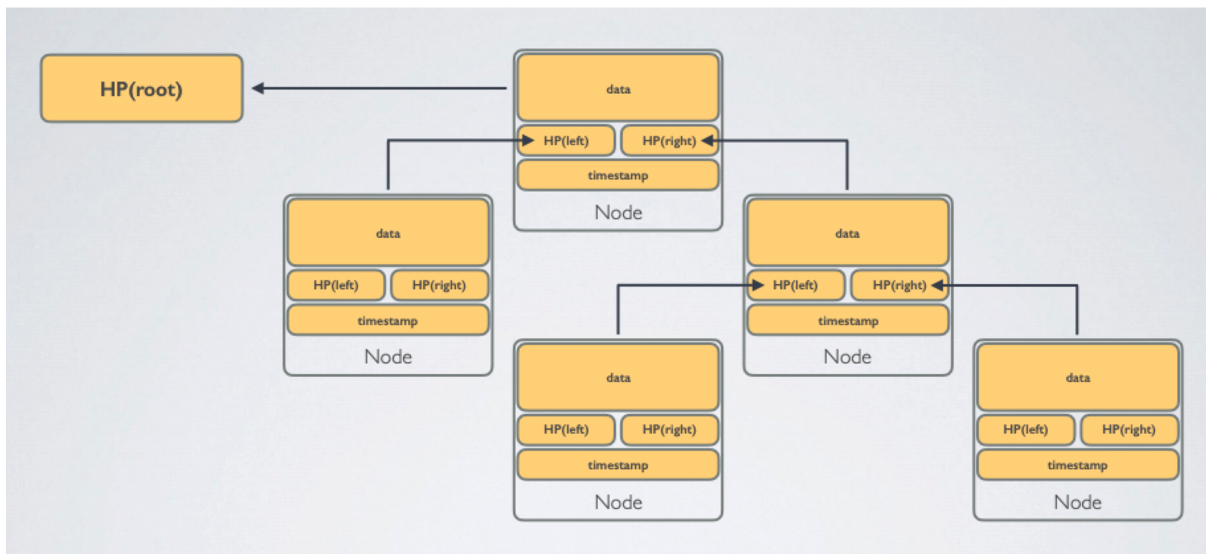

Figure 2: Merkle Tree Hashing (Gupta)

The logical next question is *who* connects the blocks? The answer is network members known as miners. When a block is added to the blockchain, it is called *mining.* Miners are integral to the blockchain system because they review transactions, make sure they are valid, and actually construct the blocks that make up the chain. During the first implementations, miners could be individuals or small groups of people with relatively limited computing power, but it is generally not this way anymore (Wang).

Every block on a blockchain is created by a miner and contains transactions for that chain. Before that block is added, it is confirmed that none of the transactions conflict with transactions already on the chain (Zheng et al. 2018). This is done to ensure that the transactions stay valid so that the same house is not sold twice or the same cryptocurrency is not sent to two different recipients. Unlike a physical dollar bill for example, a cryptocurrency coin could theoretically be in two places at once without these checks.

As opposed to certain transactions being assigned to certain miners, miners compete to create blocks with transactions that may overlap the blocks that other miners are attempting to create at the same time. However, only one of the blocks will be added to the blockchain and this is the block that has the highest proof of work (Zheng et al. 2017). Once the block is added, any other blocks in construction by other miners with overlapping transactions are rendered useless. It is an all or nothing race.

To create a block, a miner groups together a number of transactions that are released by users of the blockchain. The miner then works to turn this group of transactions into a block by first checking that each of the transactions is valid and non-conflicting with the current blockchain. Next, the miner must solve what is essentially an extremely difficult mathematical puzzle. They try to find a nonce, this is often a 64-bit number, that solves the root hash and satisfies the proof of work required. If they are lucky, they will find a nonce that works and the block is then solved (Zheng et al. 2018). Once this happens, the completed block is shared with the other miners. The miner with the most proof of work is chosen and their block is officially added to the blockchain (Wang).

Now, miners do not do this work without reward for successfully adding blocks. Once a miner has successfully completed a block that is added to the blockchain, they are typically

given a monetary reward in the form of cryptocurrency (Zheng et al. 2018). For the Bitcoin network, the reward is in BTC and for the Ethereum network the reward is in ETH. The size of this reward typically decreases over the age of the network to encourage miners to participate at the genesis of the blockchain (Wang).

It was previously noted that modern day miners are no longer individuals or small groups and this is due to the extremely high mining difficulty of the current blockchain systems. The point of mining difficulty is to be able to control the number of blocks being added to the blockchain over a period of time. Since a high mining difficulty means that a nonce is very difficult to compute, most miners that can do this are massive data centers devoted to mining (Wang). It would be virtually impossible for an individual miner to solve a nonce and establish a block before it was done by a data center with thousands of times the computing power.

Bitcoin mining, for example, has become so difficult that there are special ASIC computing chips made specifically for Bitcoin mining. Without this special equipment and large numbers of computers, trying to mine Bitcoin would cost more in electricity than it would earn in Bitcoin (Taylor). One way in which individuals can mine is to join a "mining pool" where they join with an incredibly large and often global group of people to pool their computing power (Lewenberg). The downside here is that there are so many members in these mining pools that the BTC reward per person for a successful block is relatively small. The reward is directly proportional to the share of computing power they provide.

2.1.2 History

The publication of the Bitcoin whitepaper is generally regarded as the start of blockchain, though it is important to note that there were several predecessors built on much of the same technology. Bit gold, created by Nick Szabo, was a precursor cryptocurrency to Bitcoin that also used a proof of work mechanism (Szabo). While Bit gold was created in 2005, Bitcoin was not released until 2009. *Bitcoin: A Peer to Peer Electronic Cash System,* written by the mysterious Satoshi Nakamoto, described a trustless platform to send electronic money back and forth (Nakamoto). The electronic money was called Bitcoin (abbreviated

BTC) and the system it was built upon was called blockchain. At the time Bitcoin was released, most people did not understand the difference between BTC and blockchain, or that the underlying technology of blockchain could be used for assets other than BTC (Nakamoto).

For the next five years, Bitcoin grew rapidly in popularity. Just a year after its release, a previously unknown vulnerability was taken advantage of to create billions of unauthorized Bitcoins. In just a few hours, the vulnerability was fixed and a fork was created for the new protocol. The blocks on the new fork are verified and added using the new protocol. One of the earliest Bitcoin transactions was 10,000 BTC for two pizzas (McCall). As of today, March 3rd of 2021, this amount would be worth $509,998,000. By 2012, Bitcoin was widely discussed and even the topic of several network television show episodes.

The following year, 2013 marked the beginning of serious legal troubles for many Bitcoin exchanges. The volume of bitcoin transactions had reached a large enough volume at this point to attract the attention of government financial regulators (Marr). In May of 2013, U.S. government officials confiscated many of the accounts of infamous exchange Mt. Gox for failing to properly register as a money handler (Chohan). While the Bitcoin system itself is completely separate from any fiat currencies like the U.S. dollar, the ability to *buy* BTC using USD made exchanges the target of financial law regulations. It was in the same year that a Federal Judge ruled that Bitcoins would be considered as a "currency" by U.S. law and as such, were under the jurisdiction of the U.S. court (Tricchinelli). Many believed that this idea was the antithesis of the decentralized and open nature of the blockchain.

Perhaps the most famous of the legal disputes involving Bitcoin, the website *Silk Road* was shut down by the FBI in October of 2013 (Christin). The owner of the website, Ross Ulbricht, was arrested and sentenced to a double life sentence in prison. The severity of this sentence was due to the transactions facilitated on Silk Road. It was a black market on the dark web that could be accessed through the Tor browser. Most of the products offered were illegal drugs such as cocaine and psychedelics. However, it should also be noted that child pornography and even assassinations were available on Silk Road (Christin).

In 2015, the first in-person Bitcoin exchange was legally opened in New York City. Individuals could come here to buy Bitcoins (Kirby). At its opening, the exchange was incredibly popular. However, the exchange quickly lost popularity and money until it was shut down (Kirby). Even in 2021, Bitcoin exchanges are frequently shut down for lack of popularity or legal troubles (Kirby).

One of the initial programmers of Bitcoin, Vitalik Buterin, wanted a more flexible blockchain for his new ideas (Ethereum). The Bitcoin community was unwilling to consider his ideas he proposed, so he decided to create a new Blockchain network called Ethereum. He created Ethereum with the intention of being able to support non-currency transactions (Buterin 2013). Buterin recognized the immense potential for blockchain to be used for other assets like loans or securities. The Ethereum network uses smart contracts that initiate and record transactions when criteria are met (Zheng et al. 2019). These smart contracts can be programmed using the Ethereum language Solidity ("Solidity Programming Language"). Today, many of the new blockchain applications are programmed using Solidity and tested on Ethereum test nets.

2.1.3 Cryptocurrency

Cryptocurrency is essentially a digital form of money, or asset (Narayanan). This is notably different from a *fiat* currency, which is government-issued money like the US-dollar. Where fiat currencies are centralized, cryptocurrencies are created using blockchain technology to be decentralized. Since there is no centralized authority to verify transactions, the verification must be done within the blockchain network using cryptographic algorithms. Most often this is a public key algorithm in which the receiver of the coins uses a private key to unlock the public key of the person who sent the coins. The terms cryptocurrency and token are often used interchangeably, but it is usually the case that a token refers to a digital asset built on a blockchain whereas cryptocurrencies are the native currency for their own blockchain (Narayanan).

In addition to anonymizing and verifying transactions, keys are also used to securely store cryptocurrency. Generally, a user will store their cryptocurrency in a wallet of some sort. When the wallet is made, the user creates a private key to secure the wallet so that

access can only be gained by them (Davis). While secure, this system comes with its downsides. If a private key is lost, the user will never be able to access their cryptocurrency again. Unlike a bank account, the password cannot be reset. In addition, due to the anonymized nature of the blockchain, there is no other way to prove that a certain amount of cryptocurrency belongs to you, let alone access it, without the correct private key (Narayanan). In fact, there is a large amount of cryptocurrency that is most likely unreachable forever due to users forgetting to record their keys.

Even within cryptocurrency wallets, all requiring private keys, there are differing levels of security. Many users store their coins in software wallets on the internet for convenience (Davis). This allows the user to activate the wallet directly on their browser to buy or sell cryptocurrency or to develop on a blockchain. Unfortunately, this form of storage is vulnerable to security breaches and theft.

It is also possible to store keys in a physical wallet - either on a USB or even on paper. The advantage here is that the information is not stored online, so it is more secure. The downside is that it becomes much easier to lose keys due to damage or accidents. However, in almost all instances, using a wallet involves some level of trust in the creator of the wallet not to share or steal a user's private key.

When a user sends cryptocurrency to another user, there is a transaction on the blockchain that states that the ownership of the coins has gone from the sender's address to the receiver's address. There is no actual exchange in the traditional sense. Once all of the keys have been verified and the transaction has been recorded, the wallet balances of the parties involved are altered at the same time. This guarantees that the cryptocurrency is not double-spent (Narayanan).

Today, some of the most popular cryptocurrencies include Bitcoin, Ethereum, and Litecoin. Due to the Bitcoin explosion, it is even possible to use BTC to purchase real world items such as products on Etsy marketplace or Overstock.com ("Bitcoin").

So, how does one buy cryptocurrency? The most popular way to buy cryptocurrency is using a cryptocurrency exchange. At an exchange, a user can trade fiat money or other

cryptocurrencies for the cryptocurrency they want. This is usually facilitated by a wallet that is already on the user's browser. For example, User A may go on a coin exchange, put in their debit card information, and trade a set amount of fiat currency ($) for 1 ETH. User B, may also go on the coin exchange, sync up their wallet, and trade a set amount of BTC for 1 ETH. However, it must be noted that different countries and states have different regulations on buying and selling cryptocurrency. In New York state for example, you cannot buy cryptocurrency from an exchange that does not have a BitLicense ("Virtual Currency"). Oftentimes these government licenses are time consuming and costly to get.

It is also possible to use an exchange to sell cryptocurrency for fiat currency. This is partially due to the fact that exchanges, somewhat like banks, handle a large number of transactions everyday and hold large amounts of crypto and fiat currencies. As was mentioned previously, the Mt. Gox exchange went bankrupt because huge volumes of cryptocurrency were stolen from the exchanged managed funds (Chohan). This is one major reason for individuals to keep their cryptocurrency in person wallets as opposed to in an exchange like Mt. Gox or Coinbase ("Coinbase").

Due to the tighter and tighter restrictions put on exchanges by regulatory bodies, the anonymity of cryptocurrency is becoming a thing of the past. After the Silk Road was shut down, governments realized that the anonymity of virtual currency was a threat to national security. To combat this, exchanges are now required to verify the buyer before any cryptocurrency can be sold ("Coinbase"). The higher the volume of cryptocurrency, the more verification is necessary.

Some countries have gone to the furthest extreme and made any transactions involving cryptocurrencies completely illegal. These countries include Algeria, Bolivia, Morocco, Nepal, Pakistan, and Vietnam. On the other end of the spectrum are countries that have made an effort to keep their cryptocurrency regulation relaxed such as the Cayman Islands, Spain, Belarus, and Luxemburg (Goitom). This is often done in an effort to bring the profits of blockchain technology into their economies.

2.1.4 Smart Contracts

A smart contract is an automated agreement written in code and stored on the blockchain to make it unchangeable and public. It is automated in that it is machine-executable code that is triggered when the conditions of the contract are met ("Introduction to Smart Contracts"). As was mentioned in Section 2.1.2, the ability for users to create and execute their own smart contracts on the blockchain was one of Buterin's main motivations in creating Ethereum. Smart contracts on the Ethereum blockchain are written in Solidity - a Turing complete programming language created for exactly this purpose ("Solidity Programming Language").

The clauses of a traditional contract can be encoded in a smart contract using if-else style logic. These clauses are agreed upon by all parties and the smart contract is then created (Zheng et al. 2019). Due to the automated nature of smart contracts, there is no need for an external enforcer of the contract. To show the importance of this feature, imagine that Veronica and Julian have a traditional book illustration contract. Julian is publishing a book, but needs several illustrations for the inside. Veronica agrees to create the illustrations for $X and this agreement is written out in the paper contract. They could use an escrow service, but it is far too expensive and slow for their budget and timeline. As was agreed, Veronica creates the illustrations that are put in the published book, but has not been paid by Julian. What are Veronica's choices?

Since she and Julian have a traditional contract, she would need to hire a lawyer and possibly bring Julian to court to enforce the conditions of the contract. Not only is this too expensive and time consuming for all parties, it is still possible that the court could side with Julian. If Veronica and Julian had used a smart contract instead, Julian wouldn't have been able to avoid paying Veronica. Upon completion of the illustrations, the smart contract would register this and automatically transfer the funds from Julian to Veronica.

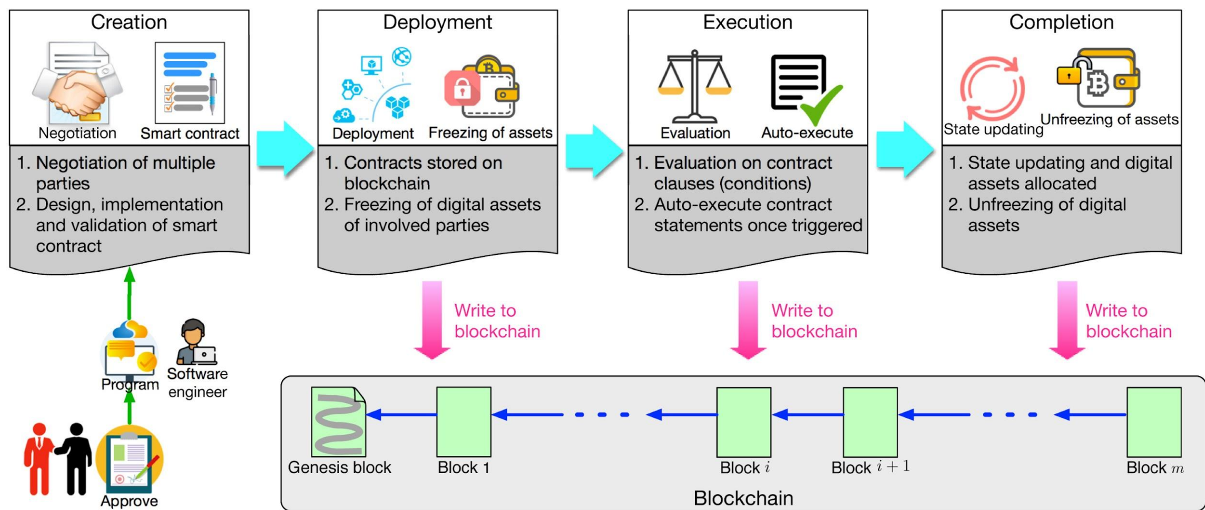

Figure 3: Smart Contract Lifecycle (Zheng et al. 2019)

The stages of smart contract creation and implementation are illustrated in Figure 3 above. The smart contract is created to reflect the needs of all those involved and then coded in Solidity. Once the contract is finalized, it is deployed to the blockchain and can no longer be amended. At this stage, it is important to note that there is a cost. Everytime a contract is deployed, the programmer must pay a certain amount, which is called "gas" ("Introduction to Smart Contracts"). The cost of this is dependent upon the gas price and complexity of the contract. While this cost has historically been small (equivalent to a few USD), at the time of writing, the high price of ETH makes launching a contract much more expensive.

The good news is that contracts can - and should - first be launched on a test net such as Rinkeby or Ropsten ("Introduction to Smart Contracts"). Deploying on these test nets is free in that doing so only requires free test net coins. Once the contract is deployed, there is also a small cost associated with any transaction that takes place through the contract. As a rule of thumb, any process that requires work from the blockchain will have a proportionate cost ("Introduction to Smart Contracts").

The execution phase of the smart contract refers to the period in which the smart contract is waiting for someone to invoke it. Once this happens, the correct functions within the contract are triggered and transactions are executed. These transactions are then validated and recorded. Finally, the completion stage involves updating states to reflect the current post-transaction balances (Zheng et al. 2019).

The advantages of smart contracts are immense and span a wide variety of industries. In general, they eliminate middlemen, create a trustless environment, reduce cost, increase security, and at the same time guarantee transparency and immutability of transactions (Lipton). In theory, it isn't possible for one party to cheat another because the contract is automatically executed when the conditions are met and that transaction is then permanently recorded on the blockchain.

Having discussed the benefits of smart contracts, it is also important to understand their existing weaknesses. Smart contracts struggle when it comes to coding ambiguity, real world actions, and physical assets (Lipton). The wording of paper contracts is often such that a level of ambiguity is present within the conditions. In the logical structure of smart contracts there is no room for interpretation like this. In addition, it is often the case that conditions of contracts are dependent upon real world actions and the verification of the completion of these actions is tricky. This is most often done using Oracles - sources for verified information ("Oracles"). Unfortunately, the validity of this information can vary. The more "objective" the information, the better Oracles do. If a smart contract condition is triggered by the S&P 500 reaching a certain valuation it is reasonably reliable to pull this information from an Oracle. However, it would be difficult to use an Oracle to verify that an action such as renovating a house had been completed ("Oracles"). In addition, smart contracts can contain errors and lack a formal system of enforcement (such as the legal system for traditional contracts). This leads us to the difficulty of representing real-world assets (Lipton).

For example, imagine Brianna and Mark have used a smart contract for a car sale. The most basic contract would state that Brianna gives the car to Mark and the funds are automatically transferred. But how does the smart contract validate that the car is physically in Mark's custody? If Mark needs to validate this then he could say he never received the car so that he didn't have to pay. If Brianna needs to validate this then she could say she gave Mark the car when she did not and charge him for it. The bottom line is that it is difficult to maintain a trustless environment when handling physical assets.

2.1.5 Decentralized Applications (DApps)

A DApp is similar to what one would think of as a normal "app" in many respects. It can have a user interface (UI) much like an app on a smartphone or computer. On the front end, a DApp will often look the same as the apps we are familiar with. On the back end, a DApp is built using smart contracts on a blockchain (i.e., *decentralized* platform). A DApp can pull from new smart contracts that are created by the developer specifically for that DApp, or from contracts already deployed on the Ethereum blockchain. The idea is that the data in a DApp comes from the blockchain as opposed to servers ("Introduction to DApps").

It is common for DApps to have their own coin that is used to interact with that DApp. Consider Uniswap, one of the most popular DApps in the world ("Uniswap"). It is a cryptocurrency exchange that uses what is called an "automated liquidity protocol" to regulate the exchange rate between cryptocurrencies. As it is built on the Ethereum blockchain, it is able to deal with ERC-20 tokens and Ethereum wallets. It has more than $3 billion in cryptocurrency assets within its liquidity pools ("Uniswap").

Users are able to interact with Uniswap via the UI while smart contracts on the back end execute cryptocurrency transactions. Users use UNI tokens, the Uniswap token, to interact with the DApp in various ways ("Uniswap"). UNI tokens can be exchanged for other cryptocurrency tokens, held on to because their value is proportional to the value of Uniswap, or utilized to vote on Uniswap issues much like a shareholder. UNI tokens were distributed with an Initial Coin Offering (ICO) in which tokens were given to Uniswap community members on a certain schedule. Exactly 1 billion UNI tokens were created and distributed over the course of four years ("Uniswap").

This DApp structure has allowed Uniswap to grow at an unprecedented level, but there are downsides. The primary consequence of running an application over the blockchain is that transactions can be wildly expensive - monetarily and computationally. As congestion on the Ethereum blockchain rises, so does the gas price of launching a contract or implementing simple transactions. In addition, massive overhead makes scaling very difficult. The official Ethereum website states that, "A back-of-the-envelope calculation puts

the overhead at something like 1,000,000x that of standard computation currently" ("Introduction to DApps").

Also, the need for constant code maintenance and the problem of deprecated functionality present in regular apps carries over to DApps. The immutability of information on the blockchain can make it difficult to amend or update code. If a developer wants to change a smart contract, they will need to deploy an entirely new and updated contract to the blockchain. This is inconvenient and expensive.

2.1.6 Ethereum

Ethereum was designed to utilize blockchain technology for more than just monetary transactions (Buterin 2013). It was designed for developers to be able to create their own applications using smart contracts. These include name registration, colored coins, CryptoKitties, property transactions, and much more (Cryptokitties).

Ethereum runs on its own cryptocurrency - ETH. It is primarily used to pay gas fees for launching smart contracts or carrying out transactions. Ethereum sets a maximum number of steps that can be taken to carry out the transactions on the block - the gas limit. In addition to this, there is the start gas, which is the amount of gas that a user sends for the transaction to be processed. If the transaction ends up using less gas than anticipated, the difference is refunded back to the user (Wackerow). This system is shown in figure 4 below.

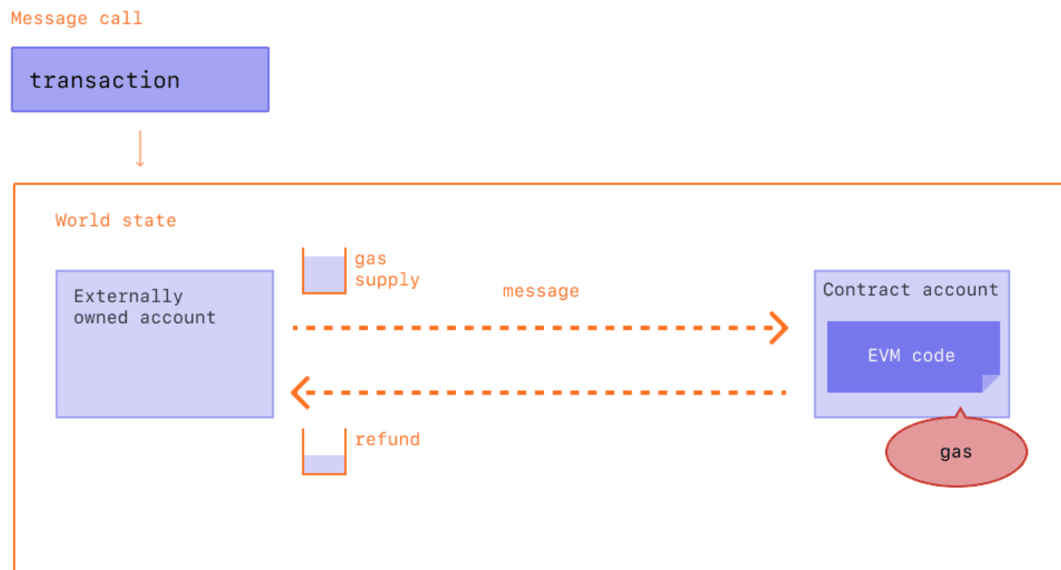

Figure 4: Gas Use From Transactions (Wackerow)

Gas is related to ETH in that there is a gas price in ETH for each unit of gas. Technically, the gas price is listed in Gwei which is equivalent to 10^{-9} ETH (Wackerow). The purpose of gas fees is to regulate usage of the Ethereum blockchain. The limit makes sure that there are no loops in the code that go on forever wasting space and resources. The cost makes sure that developers don't spam the blockchain with transactions. Not only would this be computationally expensive and therefore harmful to the environment, it would also overwhelm the blockchain to the extent that other developers couldn't use it.

Another feature of Ethereum is that it was built with quick development and deployment in mind. This is why Buterin built it as a blockchain with a Turing-complete development language, Solidity ("Solidity Programming Language"). This allows anyone to create their own DApps with specialized smart contracts for any kind of functionality they want. Solidity was structured in such a way that developers could get a baseline level of functionality with relatively few lines of code. In addition, of course, Solidity can simply and easily interact with the blockchain to carry out transactions ("Solidity Programming Language"). The philosophy behind Ethereum is "simplicity, universality, modularity, agility, non-discrimination, and non-censorship" (Buterin 2013).

To interact with the Ethereum blockchain, you need an account. There are two types of accounts: externally owned accounts (EOAs) and contract accounts (Mi). An EOA is what is generally thought of when one references Ethereum accounts. An EOA is controlled by a key pair such that a user with the correct keys can create transactions or send ETH from that account. Metamask accounts, for example, are a common type of EOA (Davis). Contract accounts are controlled by the code inside of the contract such that whenever the account is interacted with, the code is activated. The contract is in control of the ETH in the account. Contract accounts are essentially smart contracts that hold ETH within them. It is then possible for this ETH to be transferred from the contract to an EOA (Mi).

2.1.7 ERC-20 Tokens

The ERC-20 is a token standard that aims to set certain requirements for fungible tokens (Buterin 2015). This means that users who trade the token or developers who build applications using the token can be assured that certain features will be present. It is also only applicable to tokens of the Ethereum blockchain. A fungible token, in contrast to a non-fungible token (Section 2.1.8), is a token type where any token can be swapped with another. In other words, the tokens are not unique in value or any other feature. One simple example of this is the token DAI - person A’s 1 DAI is equivalent in every way to person B’s 1 DAI (“An Unbiased Global Financial System”).

A token that implements the ERC-20 standard is a bit like a class inheriting from a parent class in Java. The ERC-20 API is the parent class and the smart contract defining the token is the child class. Any ERC-20 token contract is guaranteed to have certain functions and events and may have other optional functions. The three optional functions present in the ERC-20 are `name()`, `symbol()`, and `decimals()` (Buterin 2015). The `name()` function returns the name of the token in string form. The `symbol()` function returns the symbol of the token in string form. This is usually a three letter string such as “ETH” for the Ethereum token. The `decimals()` function returns the number of decimal places used by the token (Buterin 2015).

The two events that an ERC-20 token contains are `Transfer(from_, to_, value)` and `Approval(owner, spender, value)`. These events are triggered when the

`transfer()` or `approve()` functions are executed and notify the developer of this. They return a boolean variable for the success or failure of the operation. While this seems small, it is essential to have events listening for these actions to know what is going on with the token. In addition, other logic in the smart contract can be triggered by these events (Buterin 2015).

In addition to events and optional functions, there are six required functions (Buerin). The `totalSupply()` function returns the total number of tokens to be made. Calling this function tells a developer how many tokens are out in the world. The `balanceOf(address)` function returns the total token balance of the user with the address specified. The `approve(_spender, value)` function checks whether a receiver is allowed to receive the tokens and makes sure that there is no fraud or counterfeiting. This function triggers an `Approval(owner, spender, value)` event. After this, there is the `transfer(_to, _value)` function which sends the specified number of tokens from the owner to the recipient. This function triggers a `Transfer(from_, to_, value)` event. There is also a `transferFrom(_from, _to, _value)` function which essentially allows the contract to automate sending tokens to someone for you. This function also triggers a `Transfer(from_, to_, value)` event. Lastly, the `allowance(_owner, _spender)` function checks how much the spender can take from the owner. Part of this requires checking that the owner has enough tokens for a transaction (Buterin 2015).

The ERC-20 token standard has been widely adopted and promoted by the Ethereum Foundation for several reasons. The common standard avoids blockchain chaos from incompatibility. Any ERC-20 token will have those previously mentioned functions in common with all other ERC-20 tokens. This means that a developer knows what to expect and can create applications that rely on the functionality of the ERC-20. Also, the token standard reduces the barrier to entry for novice developers to create tokens on the Ethereum blockchain. To the Ethereum foundation, the more awareness and development on their blockchain the better. Also, any token that is ERC-20 compliant can still contain as much additional functionality as the developer likes (Yilmaz).

To some developers, this reduced barrier to entry is a consequence as opposed to a reward. Since the ERC-20 token standard allows for streamlined creation of tokens, there are thousands of tokens flooding the blockchain (Yilmaz). Many of these tokens are created by

novice developers and are not used or adopted by others. There are also a number of disingenuous token developers that now have an even easier time creating tokens to scam people (Yilmaz). In addition to this problem, there is a major flaw in the structure of the ERC-20.

There is a bug in the way the transfer function is defined that has caused many people to lose cryptocurrency in transfer. If an individual wants to transfer tokens to an account that is managed by contract code (as opposed to a regular individual account), the owner of the contract account won't receive a notification about the transfer. Due to these weaknesses, there are other fungible token standards such as ERC-223 that have met with varying levels of adoption (Buterin 2015) (Yilmaz) (Dexaran).

2.1.8 Non-fungible Tokens (NFTs)

The ERC-20 token standard described above only applies to fungible tokens. These are tokens that are interchangeable much like one dollar bills for example. There is no difference in value between individual tokens. In addition, much like ERC-20, there are a few main NFT token standards (Buterin 2015). NFTs are tokens where each individual token is unique in value or attributes and therefore cannot be swapped interchangeably for another token. Due to their structure, NFTs are well suited for tokenizing unique digital or physical assets like in-game items, properties, or art.

The most widely used token standard for NFTs is ERC-721. This standard contains functions and events similar to that of the ERC-20, except that they are specific to the non-fungible structure of the coins being created (Entriken). For example, based on the unique ID of the token, it is possible to generate an image specific to that token using the ERC-721 Metadata JSON Schema. This is what allows users to see the unique visual characteristics of what the token represents. The standard guarantees that the ownership of every token is individually tracked and that every token can be differentiated from the others. It also allows tracking and transferring of tokens.

If we consider the example of a piece of property, say a single family home, we could create a smart contract based on the ERC-721 standard to represent it. The smart contract

would generate a unique token for that house and the metadata could include photos and other important information such as the address, number of bedrooms, number of floors, year built, etc. We could then track the ownership of the house token or transfer it to another user. We could do the same for a piece of art or even a loan (Entriken).

However, there are some limitations to both the ERC-20 and ERC-721. One such limitation is that they are designed for contracts that only specify one token type. With these standards, it isn't possible to have a smart contract that deploys both fungible and non-fungible tokens. It's also impossible to create semi-fungible tokens which can be thought of as similar to a gift card or seat in a theater. They are fungible until used or redeemed, and then become non-fungible (Yilmaz) (Entriken).

The ERC-1155 standard allows for the creation of multiple tokens within one contract (Radomski). It can also be used to create semi-fungible tokens. Smart contracts implementing ERC-1155 can also transfer tokens of different types all at once, allowing the user to save on transaction costs. It also allows for the store of metadata unique to each token. While it is more versatile, the adoption of ERC-1155 is not as widespread as its predecessors.

NFTs as a whole, however, have taken the blockchain world by storm. Some of the most successful NFT applications to date have been digital art and in-game assets ("Gods Unchained"). Part of the reason for their success is that these applications did not face the hurdle of creating a system for tokens to represent real, *physical* assets. The assets are digital and thus it is possible to completely capture them via a token.

One particularly unusual and noteable use of Ethereum blockchain is the creation of CryptoKitties. These are collectable, virtual cats with different characteristics and rarities. In 2017, a first generation CryptoKittie was sold for around $120,000 (CryptoKitties). Hashmasks are collectible pieces of digital art created using the ERC-721 standard. The initial sale brought in approximately $16 million worth of ETH (Suum Cuique Labs GmbH). To many in the art world, this came as a game-changing shock. Gods Unchained is a game involving trading cards, where each card is represented by an NFT in the game ("Gods Unchained"). The game is widely played and has received millions in funding. Given any

possible application of blockchain, there is undoubtedly a company or programmer that has implemented a version of it.

2.2 The Real Estate Market

The real estate industry in the U.S. is worth trillions of dollars. In 2020 alone, U.S. housing increased in value by $2.5 trillion (Richardson). This is a valuable market and it continues to increase in value, despite a global pandemic. We spend most of our time in buildings and those buildings are pieces of property with transaction records, owners, and monetary worth. Every transaction that a building goes through is handled by a series of players in the real estate market. We will begin by doing an overview of the basic structure of the market in the U.S. From there, we'll go over the roles of the major players and intermediaries in any real estate transaction and then discuss the areas within the real estate industry that are a potentially promising fit for blockchain technology. Lastly, we'll look at what current blockchain-based companies are doing in some of these areas and isolate existing open opportunities for growth. These openings will be the basis for the DApp created in this dissertation.

2.2.1 Basic Structure in the United States

The real estate market and related industries are responsible for approximately 30% of the U.S. GDP - this is a major market (Glickman "Introduction"). To understand the market and how it is open to blockchain technology, we will start with the basic structure in the U.S.

There are two main types of real estate: residential and commercial. Residential real estate refers to family homes and apartments. This type generally refers to properties that people live in. On the other hand, commercial real estate refers to almost everything else. This type encompases company buildings, offices, shopping malls, hospitals, hotels, etc. (Glickman 1-4)

The residential segment can be broken up into two main subsegments: single-family homes and multi-family properties (Glickman 1-4). Single-family homes are generally under three stories, often contain a garden space, and are usually occupied by the owner of the home and/or the owner's family. This is the property type that most people imagine when they think of real-estate - a basic house. Multi-family properties are broken down by the

number of floors into low, mid, and high-rise buildings. These can be apartment buildings, condominiums, or communal style living in which many families live in a connected building and share a garden. The units within these buildings can be rented out (generally apartments) or lived in by the owner (generally condominiums). Typically, the more urban the area, the higher-rise the properties. In major cities, it is often the case that the residential real estate market is mainly composed of units in these high rise apartment buildings (Glickman 1-4).

The commercial segment has a large number of subsegments - one of which is office buildings, which can be further broken down by their quality rating and location (Glickman 1-4). Next, we have retail buildings such as shopping malls. These can have many different indoor and outdoor layouts. There are also hotel properties, which make up a surprisingly large number of commercial properties considering how niche the subsegment seems. Another major subsegment of the commercial market is industrial properties. This includes factories, warehouses, and general purpose industrial plants. These are often large plots of land with specific location requirements, such as close proximity to highway entrances (Glickman 1-4).

Given these definitions, it is important to understand the distinction between real property and personal property. Personal property refers to things that are owned by an individual that are movable by that person. This includes things like books, art, and clothing. The majority of the NFT applications discussed in the above section are an attempt at representing personal property in the blockchain. On the other hand, real property refers to land and buildings that are owned by an individual. These are assets that cannot be moved by that person. As will be addressed later, this form of property presents unique opportunities and challenges for blockchain application. The real estate market, of course, primarily deals with transactions of real property (Glickman 9).

The transactions in this market are not as simple as transfers of ownership. This is due to the fact that there are many different forms of property ownership and use. Each one comes with its own rights, responsibilities, and contracts. There are two main buckets: direct ownership and indirect ownership (Glickman 9).

Direct ownership of a property is either Single Proprietor, Tenants in Common, or Joint Tenants (Glickman 9-13). Single Proprietor properties are owned by an individual and all rights and responsibilities fall on them. After they die, the property ownership is transferred to heirs unless otherwise specified. This also means that the individual may decide to sell, rent, finance, etc. the property as they wish. Tenants in Common properties are owned by multiple people each with a percentage of the property. This means that decisions regarding the property must be agreed upon by all owners or by a designated third party. Joint Tenant (typically with rights of survivorship) properties are generally owned by married couples. In this case, the property is completely owned by both parties such that if one party passes away, the property is still owned by the other party without it passing into an estate etc. These cases are fairly straightforward. However, it becomes a bit more complicated when dealing with indirect ownership (Glickman 9-13).

Indirect ownership of a property is generally one of a few categories: Partnerships, Corporations, LLCs, REITs, REMICs, SIVs, and SPEs (Glickman 15). Partnerships are essentially a legally recognized group of owners of a property. This is often thought of as an inbetween from direct ownership to indirect ownership as the legal entity of the partnership is created for the purpose of property ownership by a group of people. Corporations are companies that are owned by their shareholders (i.e., those that invest in the company, elect a leading body, etc.). This means that the corporation has a legal responsibility to those shareholders. When a corporation owns a property, this is indirect ownership by the shareholders and those financially involved. LLCs are Limited Liability Companies are a fusion of the previous two in that they have the tax advantages of a partnership while protecting its members from company liability ("Limited Liability"). REITs are Real Estate Investment Trusts are very similar to Corporations with the exception that their investments are constrained to real estate ventures. They were created for the purpose of allowing the average person to invest in real estate without massive amounts of capital. REMICs are Real Estate Mortgage Investment Corporations. As the name suggests, these companies invest in mortgages and mortgage securities. Similar to REMICs are SIVs (Special Investment Vehicles), which essentially borrow money at low interest and invest it in mortgage-backed securities with a higher return. Lastly, SPEs are Special-Purpose Entities which are generally formed to legally separate a property from the other assets of the owner for a variety of financial and liability purposes (Glickman 15).

Everything that has been laid out thus far is the general structure of the real estate market and real estate finance. The next section will focus on the major players in real estate development, sales, etc. Understanding the middlemen involved in all of these real estate transactions is necessary to isolate where blockchain technology can be effectively applied.

2.2.2 Industry Participants

The real estate industry in the U.S. and abroad is known for its overwhelming abundance of middlemen at the center of all transactions (Glickman 17). Before analyzing the roles of everyone involved, we need to define the main processes they are involved in.

Property development involves choosing a space, obtaining government approval, planning the project, and all logistics before the property is built. The construction stage is when the property is actually built. Once a property is built, it needs to be managed. Property management involves upkeep, repairs, property finances, and security. Assuming the property will not be lived in by the creator, next comes property leasing or property brokerage. Property leasing involves finding tenants to rent the property, while property brokerage involves finding someone to purchase the property. A property may be leased out after it is built or it may be sold and lived in or leased by the new owner - there are a number of possibilities. There may also be asset management of the property. This process involves analyzing the worth of the property relative to the neighborhood and market, analyzing possibilities to increase value if applicable, and deciding the best time to put it on the market (Glickman 18).

Within each of these processes, there are a variety of professionals involved, many of whom add high expenses to the cost of a real estate transaction. Firstly, there are those involved in the planning and building of the property. Before the build can begin, it needs to be funded. Unless this is paid for out of pocket, funding may originate from bank loans. Construction companies handle the building construction. This involves construction workers as well as subcontractors for specific tasks. Architects design and plan the building. Surveyors analyze the land of the property to determine measurements, risks, and areas to

build. Some of the more technically precise work of the build is done by engineers. Civil engineers are often brought in to make sure the structural integrity, electric, and water systems are sound. Environmental engineers provide recommendations to make the property construction and design as environmentally responsible as possible. Mechanical and electrical engineers handle a variety of specific implementations such as air systems, elevator installation, and electric system set up (Glickman 18).

Once the property has been built, the owner may work with a brokerage to put the property up for sale. They may also work with a leasing firm that hires brokers to find tenants that match the owner's criteria. The owner may also choose to have a property manager who will handle almost everything to do with the property logistics and maintenance. This may include collecting rent payments, paying workings on the property, etc. Along with the property manager, there are a large number of specialized players involved in property transactions and upkeep. Lawyers are needed to draw up sale and lease contracts along with tax documents. Accountants are hired to keep track of the finances of all transactions involving the property. In addition, a prospective buyer or investor may hire an appraiser to assess the true value of the property. In addition to a regular broker, an insurance broker is often hired to assess which insurance plans are best for the property and owner. This could involve title insurance to protect the owner from outside claims to the property. It might also involve natural disaster insurance, theft and property protection, etc. (Glickman 18).

When it comes to the purchasing of a property or investment in a property, banks are heavily involved. Unless the prospective buyer can pay out of pocket, they will require a bank provided mortgage. To obtain a mortgage, the individual's financial record is assessed by the bank. The bank then tells the individual the amount of money they are willing to provide them to facilitate the purchase of the property. The prospective buyer then goes to the real estate agent or broker with their offer and proof of mortgage approval. If the owner accepts the offer, then the buyer will put down a certain amount of money out of pocket. The property is then in escrow. In escrow, the funds are held by a third party for a period of time to make sure everything is in order with the property, seller, and buyer. When the house closes, the buyer is officially the owner and pays a predetermined monthly amount back to the bank for the mortgage. This is an oversimplification of the process, but provides an easy to understand outline (Glickman 16-17).

Another way in which a bank may introduce a claim on a property is in the case that a mortgage is defaulted on. In this case, or in the case where the property was put up as collateral to the bank for another loan, the bank may seize the property. This is another property transaction that would need to be validated and recorded (Glickman 12).

Investing in a property does not usually involve normal family homes, but often commercial real estate or housing developments. Organizations like REITs are built for individuals to be able to invest in real estate. On the other hand, investment banks or mortgage banks allow large scale investors and companies to invest in commercial property or mortgage securities (Glickman 15).

From an initial survey, it is already clear that there are a huge number of participants in the industry and an expensive chain of middlemen involved in any property transaction. In addition to high costs for potential buyers, tenants, or investors, the system is set up inefficiently. Even a straightforward home sale can take months to close due to the structure explained above. These difficulties make the real estate market an ideal area to apply blockchain technology solutions (Glickman 1-18).

2.2.3 Areas Open to Blockchain-based Technology

There are many areas within the real estate market where blockchain technology can be applied, but for our purposes we will focus on four key applications: 1) tokenizing properties, 2) tokenizing securities, 3) rental transactions, and 4) ownership and transaction recording (Smith et al.). This is by no means an exhaustive list, but for the purposes of this dissertation, they are the applications that we will focus on.

Tokenizing properties is similar to tokenizing securities in that they both involve splitting an asset into tokens - each representing a fraction of that asset (Smith et al.). In the case of properties, the asset is the property or some shell corporation representing the property, while in the case of securities, the asset is usually debt or equity. That debt could be a mortgage or even a student loan. However, for commercial properties like resorts, the

security can be equity in that business. For example, the Aspen Coin represents equity in the St. Regis Aspen Resort in Colorado (Aspen Digital Inc.).

The rental market, especially short term and vacation rentals, provides an excellent application for blockchain technology. For short term rentals, it is often a hassle to handle the logistics of passing off the keys, coordinating times, and etc. By utilizing the blockchain, it is possible to have a smart contract handle the logic of payment and property access (Smith et al.). Once the renter pays the cost of the rental, the smart contract will register this and send the door code to them. This is of course dependent upon the use of external locks. For longer term rentals, a smart contract can also securely store a security deposit, which allows for a trustless environment (Smith et al.).

Ownership and transaction recording is debatably the most important of all the blockchain applications in real estate. Most property ownership records are still kept on paper in local municipalities. This style of ledger is vulnerable to corruption, natural disaster, and human error. By storing ownership records and all other important property transactions on the blockchain, they are immune to the previous vulnerabilities (Smith et al.).

Due to limitations on time constraints and complexity, this dissertation project will focus on the latter two areas: rental transactions and ownership & transaction recording. However, tokenization in real estate would be an excellent area for future development in this field.

2.2.4 Current Companies

There are currently far more companies in the blockchain real estate sector than one might expect. Since this is a new area for development, it makes sense to highlight the work of current companies as opposed to a very short history. Here we will highlight some industry leaders in the above mentioned application areas.

Propy is a listing platform with data from the MLS where users can list their homes or search for property to buy (Karyaneva). If they decide to make an offer, the paperwork is filled out via the Propy DApp and recorded on the Ethereum blockchain. At the end, if

everything goes through, the buyer receives the property title with its own blockchain address. Since the title is on the blockchain, the buyer can be guaranteed that their ownership of the property is immutable. Propy is also compatible with existing real estate systems in that it also recorded ownership with the official government registry - all bases are covered. Since the transactions and paperwork are handled using smart contracts, the environment is trustless, fast, and is much less expensive due to the lack of middlemen.

RealBlocks is a platform for investors to diversify their real estate investing portfolios (Quarshie). They have tokenized properties on the blockchain and created a DApp marketplace for users to buy tokens representing a share of that property. This way, an investor can own property all over the world without crazy fees, in-person visits, or worrying about being entirely financially responsible if something goes wrong with a property. The RealBlocks DApp is also compatible with Metamask, which means that buying a property token is as easy as clicking a button.

Ubitquity is focused on creating a blockchain ledger of ownership for a range of assets (Wosnack). They create unique UIs on the DApp for each client and have been used to modernize government ledgers and record property ownership in a way that is not vulnerable to attacks or theft. Ubitquity primarily targets companies that want to have blockchain capabilities and works with their existing system to do so. This is called Blockchain-as-a-service (BAAS).

RealT is very similar to RealBlocks in that they allow users to buy tokens that represent a fraction of a property (McCulloch). The difference however is that Realblocks is targeted towards larger investors that want to diversify their portfolio, while RealT is more friendly to the novice investor. On their website, RealT states “A single token for RealT properties costs between $50 – $150 per token, which are the lowest investment minimums the real estate industry is able to offer. Traditional pen and paper competitors to RealT have $5,000-10,000 investment minimums.” (McCulloch) In addition, RealT pays the users’ interest in the form of stable coins. Stable coins are a cryptocurrency that act like the fiat currencies we are familiar with - one stable coin (DAI for example) is ~$1. It is also worth noting that in order to tokenize the properties, RealT creates an individual LLC for each

property and then tokenizes that LLC. This is done as a work around to government restrictions on directly tokenizing property.

Securitize is a platform that allows users to trade digital securities on the blockchain (Domingo). Securities are financial assets with some worth like equity or debt. The term digital securities refers to securities that are tokenized and therefore tradeable on the blockchain. For example, it would be possible to own and trade equity in a home. This is an investment opportunity for users and a way for sellers to raise capital and create liquidity. Of course, Securitize has their own marketplace to trade these digital securities.

The Beenest is a short-term rental platform powered by the Bee Token (Chou). This startup is the main motivation for our DApp in that it uses the blockchain to automate the short-term rental process. In other words, the Beenest is a bit like Airbnb on the blockchain. Their comprehensive UI allows users to search for and book short term stays. The rental agreement is automated through a smart contract so that both the user and the property owner can be assured the process will run smoothly. Users are also able to make payments through the system. In addition, one the user pays for the rental, the smart contract will automatically release the keycode to the door. This is a feature that is also coded into our DApp along with a full-scale tenant registry.

There are several companies working on revolutionary projects in the blockchain real-estate space. A few of these companies serve as inspiration for our DApp, and many shine a light on areas for future work. The idea for creating a comprehensive property ledger was inspired by Ubitquity and the idea of automating rental transactions was inspired by The Beenest. The main purpose of our DApp is to combine these ideas in a new and promising way. Not only will we have a property ledger, but that property ledger will be integrated with a tenancy ledger to allow us to track and change property records along with tracking the tenancies specific to each property. In addition, we will set up a payment system such that an owner can make a payment on a property and a tenant can make a rent payment. In addition, once the first payment (this could also be considered the security deposit) is paid, the keycode to the property is released to the tenant.

Chapter 3

Design

3.1 Summary of the Approach

To create our DApp, we need to design a back end and a front end. The back end design includes planning an interconnected system of smart contracts. These smart contracts will then be deployed on the blockchain this way, the data and transactions of those contracts will be stored on the blockchain. Then, instead of pulling data from a server, the front end will pull data from the blockchain and display it. It is also important that the design of the front end will allow for users of the DApp to initiate certain transactions, such as adding a property to the property registry, from the UI.

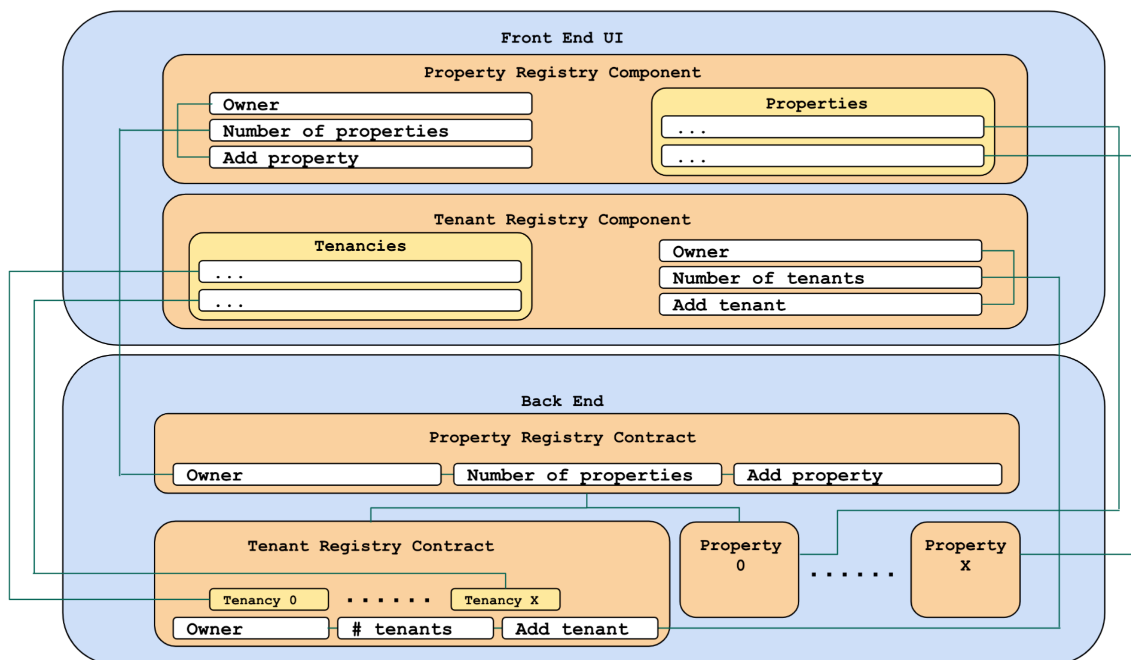

Figure 5: Basic Structure of the DApp Diagram

The figure above shows the basic structure of the DApp. In order to make the figure a bit more concise, we have left out many of the variables and functions. The general structure

of the back end is that it will be made up of three unique solidity contracts. The main contract is the Property Registry which maintains a variable for its linked Tenant Registry. The Tenant Registry is also a deployed contract. The Property Registry also maintains a list of Properties where each one is it's own deployed contract. There are additional variables and functions such as an IPFS hash for metadata and a payment system. Let's explore a bit more about how the back end will work.

To begin, as soon as the Property Registry contract is deployed to the blockchain, it automatically deploys and links to itself a Tenant Registry contract. This way, every property registry will have a corresponding tenant registry and as soon as a property is deployed from the property registry, tenancies can be added to that property via the tenant registry. So how will the properties be deployed and added to the property registry? To do this, the property registry will have a function to add properties to the list of properties it keeps. This function will take the input information and use it to deploy a property contract onto the blockchain. This covers the basic information on the interconnectivity of the back end.

From here let's look at the design of all three components in a bit more detail. The property registry contract should store the address of the corresponding tenant registry, the owner, the number of properties, and the list of properties. It should have functions to add a property, add the tenant registry (this will be called in the constructor), and get the information for a property.

The property contract should store an alias, geohash, list of owners, an IPFS metadata hash, the property registry address, the tenant registry address, and the payment pool (i.e., how much money is being stored for who). It should have functions to get the balance of the contract, update the IPFS hash, add an owner, remove an owner, send a payment, and receive a payment.

Finally, the tenant registry contract should have a tenancy structure set up inside of it and a list of those tenancies. The tenancy structure should store the tenant, property, unit number, an IPFS hash for tenancy information (such as the lease agreement), a status, start time, expiration time, an early termination fee, a keycode to the door, a price per interval (e.g., the monthly rent), the number of price intervals (e.g., 12 months), and the current

balance for the current interval. The tenant registry contract itself will also have variables for the owner, property registry address, and payment pool. Then there will be functions to add a tenancy, get the number of tenancies, get the tenancy at an index, update the status of a tenancy, change the expiration time of a tenancy, link tenancy information, get the contract balance, get the balance of a tenancy, send a payment, and receive a payment.

Now, what is the front end design and how is it connected to the back end? The general structure of the front end will mirror that of the back end. There will be a parent app component which will have two child components: the property registry and the tenant registry.

The property registry component will be displayed as a colored box on the main screen of the app. At the top of the component, it will display the address of the owner of the property registry and the number of properties in the registry. This information will be pulled from the property registry contract. From here, the component will contain submission forms for each relevant function in the property registry contract. This will include adding a property, making a payment, etc. Once the user inputs the information to the boxes and clicks submit, the inputs will be captured and forwarded to the relevant function in the deployed contract to be carried out.

Within the property registry component, there will be a subcomponent for the list of properties in the registry. Each property in the list will have its alias, location, and a boolean corresponding to whether the user is the owner of said property displayed. It will also have the image files displayed corresponding to the linked IPFS metadata hashes. From here, each property will have submission forms corresponding to the functions in the property contract. These include adding an owner, removing an owner, updating the metadata, checking your balance, sending a payment to a third party, and receiving payments that have been sent to you.

Below the property registry component will be the tenant registry component. The structure of this component will mirror the structure of the property registry component. The top of the component will display the owner of the tenant registry and the number of tenancies. Below this will be a submission form to add a tenancy. Then there will be a list of

tenancies. Each element of the list will contain submission forms to update the status of a tenancy, change the expiration time of a tenancy, link metadata, get your balance, get the balance paid for a certain tenancy, send a payment to the owners of the property of a tenancy, and receive a payment on a tenancy. These correspond to functions in the tenant registry contract.

The initial structure of the property registry contract was inspired by an open-source Consensys template for registry design on the blockchain (ConsenSysMesh). This helped to form the structure of the property registry deploying property contracts and storing their addresses. The key advantage here is that this allows each property to be its own entity on the blockchain with its own transactions while still maintaining a link to the parent registry contract. The decision to keep each tenancy as a struct within the tenancy registry as opposed to launching them as tenancy contracts was that tenancies are not their own entities. There are multiple tenancies that exist within a property. Keeping tenancies as an internal struct allowed for a streamlining of that logic and for all of the payment pool and functions to be kept within the tenant registry contract.

While this is the design plan, it is ambitious given the time constraints. As will be discussed in the implementation section, we did adhere to the structure of this design, but were unable to include all of the details we had hoped. In addition, some pieces of the front end design plan were updated to reflect a more realistic set of goals. We did this to stay consistent with the primary mission of the project having been blockchain development as opposed to a decorated UI.

Chapter 4

Implementation

4.1 Overview of the DApp

This chapter will explain how the above design was carried out in development. This will include the technologies used, challenges faced, and adaptations from the design. Section 4.2 will cover the back end of the project. Section 4.3 will detail the development and difficulties of the user interface. It is important to note that the contracts took structure from the NFT token standards, but we decided not to have the contracts extend any of them within our DApp because it didn't make sense to structure the properties as tokens themselves. The primary function of the app was in maintaining registries and managing transactions as opposed to being able to buy or sell the properties themselves.

4.2 Smart Contracts

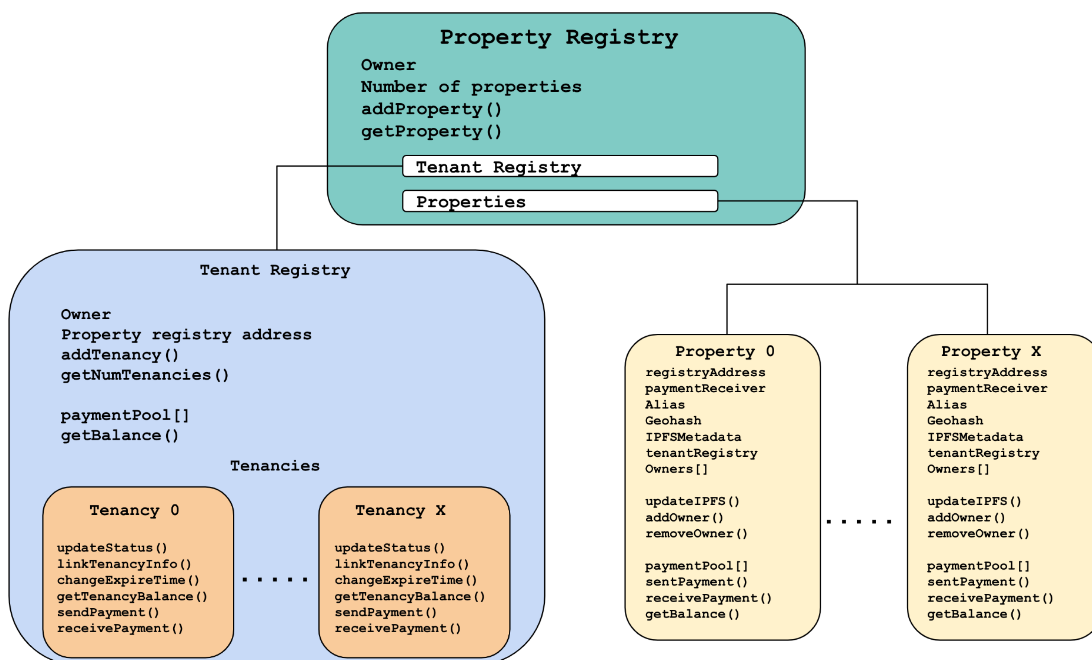

Figure 6: DApp Back End Diagram

The figure above shows the structures of the smart contracts and how they are connected. At the beginning of my development, we followed a tutorial on how to make a basic DApp (McCubbin, "The Ultimate Ethereum DApp Tutorial"). This involved creating a truffle project where truffle is a library that allows developers to make DApps and write smart contract code in Solidity. From here, we installed Ganache which runs a local blockchain complete with accounts loaded with test-ETH. Then, we loaded in a default truffle framework and deleted some of the things we didn't need.

Following this, we created PropertyRegistry.sol, Property.sol, and TenantRegistry.sol. The development of these contracts will be detailed in Sections 4.2.1-4.2.3. We began by developing inside of the truffle project, but quickly found that the debugging environment was not ideal. We then switched our contract development to Remix - an online IDE for coding and debugging smart contracts in Solidity (Remix). Once the contracts were developed and tested, the files were saved to the truffle project. The last piece was to launch the truffle project, which deployed the contracts onto the local blockchain. The UI then accessed these deployed contracts.

4.2.1 Property Registry

As can be seen in figure 6, the property registry contract is the main (or parent) contract for the project. Since the majority of the logic is placed in the child contracts, the property registry contract is streamlined and succinct. Based on the design, there were a few key pieces that were needed: property registry information, a connection to a tenant registry, and a connection to a list of properties.

At the beginning of development, we weren't sure of the best way to establish the connection to the tenant registry. We tried launching a tenant registry contract on its own, but found that there was not a straightforward way to connect the contracts if they were launched separately. After some research, we found that it was possible for smart contracts to launch other smart contracts. This provided an easy way to connect the two because any information could be passed into the child contract from the parent contract upon creation.

The next step was to decide how we would create the tenant registry inside the property registry component and how we would store the connection. We began by putting the logic to launch the tenant registry contract directly into the constructor of the property registry. Then, the address of the tenant registry that was created upon the creation of the property registry would be saved as a variable to the property registry. While this worked, we ultimately decided to create a separate function to create the tenant registry and then called that function inside of the constructor. This made the code more clear and compartmentalized the logic in a way that made more sense.

```
function addTenantRegistry()
    private
     returns (address _newTenantRegistry)
    {
        TenantRegistry newTenantRegistry = new
           TenantRegistry(msg.sender);
        tenantRegistry = newTenantRegistry;
        emit TenantRegistryAdded(newTenantRegistry);
           return (address(newTenantRegistry));
    }
```

This function begins by launching the tenant registry with an input of `msg.sender` which, in this case, corresponds to the address of the property registry. Then the tenant registry is saved to the variable `tenantRegistry`. By separating the creation of the tenant registry into its own function, we were also able to create an event for when that function was called. An event is a way to broadcast that a certain action has taken place from a smart contract. This event is triggered in the above function using the `emit` keyword. We could then check the event log on Ganache to see that the tenant registry was in fact deployed when the property registry was deployed. Lastly, the function returned the address of the tenant registry.

From the trial and error of connecting and launching the tenant registry, we had a much better idea of how to connect and launch the property contracts. We began by creating two local variables that would store the properties' information and the properties' addresses.

The properties' information was stored in a mapping called `records` which was encoded like `mapping(address => Unit) records;` This was done as a mapping as opposed to a dictionary because it is more efficient and array iteration should be avoided at all costs in a smart contract. This is because array iteration can take an unspecified amount of computing power that could use a massive amount of gas (Zheng et al. 2018). The properties' addresses are stored in an array named `keys`. It is fine for this to be an array because there is never any need to index into it in the smart contract.

```
function addProperty(string _alias, string _geoHash, string
_ipfsmetadatahash)
    public
     returns (address _newProperty, uint _keyslength)
    {
        Property newProperty = new Property(msg.sender, _alias, _geoHash,
_ipfsmetadatahash, tenantRegistry);
        keys.push(newProperty);
        records[newProperty].alias = _alias;
        records[newProperty].geoHash = _geoHash;
        records[newProperty].creator = msg.sender;
        records[newProperty].keysIndex = keys.length;
        numSpUnits++;
        emit PropertyAdded(newProperty, _alias, _geoHash);
           return (address(newProperty), keys.length);
    }
```

We then programmed an `addProperty()` function that took as inputs all of the pieces of information needed (e.g.., alias, geohash, etc.) in addition to the address of the property registry contract. This function deployed a property contract, saved the address to `keys` and the information to `records`. It then emitted a property created event and output the address of the property contract and the index of that property in `keys`. We then created a function to get a property's information from its address.

The last step was simply to create variables to store the owner of the property registry and the number of properties in the registry. The owner was saved as the address of whatever user was deploying the property registry contract. The number of properties in the registry began at zero and was incremented every time a property was added within the

`addProperty()` function. This completed the logic and functionality of the property registry contract.

4.2.2 Property

The property contract has quite a bit more logic than the property registry and contains a payment system. The first piece that we needed to program was support for multiple owners. The original plan was to simply create a local variable that was an array of owner addresses. However, some of the functions in the contract needed to be able to check whether the current user was an owner and we could not do array iteration inside of a contract. To solve this, we created a mapping called owners from a user address to a boolean that represented whether that user was an owner of the property.

```
function addOwner(address _newOwner)
public
returns (bool)
{
	require(owners[msg.sender] == true);
	owners[_newOwner] = true;
	return (owners[_newOwner]);
}

// Remove an owner from a property
function removeOwner(address _oldOwner)
public
returns (bool)
{
	require(owners[msg.sender] == true && owners[_oldOwner] ==
true );
	owners[_oldOwner] = false;
	return (owners[_oldOwner]);
}
```

Next, we created functions for a current owner to be able to add or remove an owner. They can even remove themselves as an owner. The code above makes sure that the current user is an owner and then changes the boolean value of the input user address in the owners

mapping to be the correct boolean. From here, we created local variables to store the property registry address, the alias, the geohash, the IPFS metadata hash, and the tenant registry address.

The last major piece of the property contract was to set up the payment system. The first thing we did was to create a local variable called receiver that could be assigned whenever a user invoked a send payment function to keep track of who the payment was for. Next, we created a payment pool using a mapping from user address to integer. This means that the contract would be able to see how much money was allocated to an account so that the money could then be transferred to that account.

There are four functions that make up the payment system: the fallback function, the get balance function, the send payment function, and the receive payment function. We didn't initially have a fallback function, but after researching payments in smart contracts, we realized it was very important. The fallback function serves to catch any funds that are sent to the contract without a specified function or funds that are sent incorrectly (Solidity by Example). If we didn't have a fallback function, those funds would disappear.

```
function getBalance()
public
view
returns (uint)
{
    return (address(this).balance);
}
```

The `getBalance()` function utilizes a built-in balance function. The `address(this)` code gets the address of the current contract and the `.balance` then gets the total amount of money that is currently sitting in that contract. Next comes the `sendPayment()` function.

```
function sendPayment(address _receiver, uint _amount)
public
payable
{
require(owners[msg.sender] == true);
receiver = _receiver;
require(_amount <= msg.sender.balance);
paymentpool[receiver] += _amount;
}
```

The first thing to notice is the keyword `payable` in the function header. This keyword means that this function can receive ETH when it is called. It may not seem like ETH is being sent anywhere in the code, but the keyword `payable` means that the specified amount is sent to the contract from the user's account (Solidity by Example). First we check that the user making a payment on the property is an owner of the property, then we set the variable receiver to be the receiver specified in the input. Next we check that the user has enough funds in their account to make the payment. Lastly, we register the payment deposit in the payment pool indexed to the specified receiver. We designed this function to take any receiver because this allows an owner to have the flexibility to make any kind of payment on the house to anyone (bank, contractor, etc.). Lastly, we created a `receivePayment()` function.

```
function receivePayment()
external
{
uint pmnt = paymentpool[msg.sender];
msg.sender.transfer(pmnt);
paymentpool[msg.sender] = 0;
}
```

This function uses the external keyword which means that the function should be called outside of the contract. This is the generally expected structure of receive payment functions so that any user with funds allocated to them in the contract can access them. First, the function gets the value of the amount of ETH that was deposited for the user. It then transfers that ETH to the users account and resets that users amount in the payment pool to 0.

This completed the development of the property contract. From here we moved on to what would be the most complex of the contracts - the tenant registry.

4.2.3 Tenant Registry

The first step in creating the tenant registry contract was to create a tenancy structure.

```
struct Tenancy
{
   address tenant;
   address property;
   string tenancyInfo;
   uint status;
   uint startTime;
   uint expireTime;
   uint earlyTerminationFee;
   uint keyCode;
   uint pricePerInterval;
   uint numPriceIntervals;
   uint currentBalanceForInterval;
}
```

The struct keyword tells Solidity allows the developer to essentially create their own type that stores a custom set of information. In this case, the contract will know that anything of type Tenancy contains a tenant address, property address, apartment number, info hash, status, start time, expiration time, early termination fee, keycode, price per time interval, number of time intervals, and current balance for the interval. Most of these pieces of data are self explanatory, but the last few may not be. The keycode is the code to get in the door of the unit that is being rented. The price per time interval is the rental charge per week, month, etc. The number of time intervals is how many times in total the tenant needs to pay that price (e.g., 12 for 12 months of rent). The current balance for the interval is how much the tenant has already paid towards their rent.

Next we created a local variable `tenancies` that is an array of all the tenancies in the contract. Then we created a mapping from the tenant address to their tenancy and a mapping from the tenant address to the property that their tenancy is in. These mappings would prove

useful for functions in this contract and were necessary to avoid array iteration. We also created local variables for the owner of the tenant registry contract and the address of the property registry contract that this tenant registry was for. We also needed to create a function to add a tenancy to the registry.

```
function addTenancy(address tenant, address property, uint
apartmentNumber, uint startTime, uint expireTime, uint
earlyTerminationFee, uint keyCode, uint pricePerInterval, uint
numPriceIntervals, uint currentBalanceForInterval)
public
returns (uint)
{
   uint len = tenancies.length;
   tenancies.length++;
   tenancies[len].property = property;
   tenancies[len].apartmentNumber = apartmentNumber;
   tenancies[len].tenant = tenant;
   tenancies[len].startTime = startTime;
   tenancies[len].expireTime = expireTime;
   tenancies[len].earlyTerminationFee = earlyTerminationFee;
   tenancies[len].keyCode = keyCode;
   tenancies[len].pricePerInterval = pricePerInterval;
   tenancies[len].numPriceIntervals = numPriceIntervals;
   tenancies[len].currentBalanceForInterval =
     currentBalanceForInterval;

   uint olen = tenanciesIndexedByTenant[tenant].length;
   tenanciesIndexedByTenant[tenant].length++;
   tenanciesIndexedByTenant[tenant][olen] = tenancies[len];
   tenantToProperty[tenant] = property;
   return len;
}
```

You may notice that there is no input for tenancy info. This was done on purpose because we decided that this should not be required to create a tenancy. Instead, we created a function later that could link tenancy info to a tenancy. The code of the function above adds all of the tenancy information to the tenancies array. It then creates the correct entries in the

tenanciesIndexedByTenant and tenantToProperty mappings. Finally, it returns the index in tenancies that corresponds to the tenancy that was just added.

Next, we created five functions to add, get, and remove tenancies' information. The getNumTenancies() function returns the number of tenancies in the registry by getting the length of the tenancies array. The getTenancyAt() function takes in a tenancy index and returns the most important pieces of information about the tenancy. The updateStatusOfTenancy() function allows the user to change the status of a tenancy to 0 (active) or 1 (nonactive). The changeExpireTime() function contains some more complex logic.

```
function changeExpireTime(uint tenancyIndex, uint newExpireTime)
public
{
    address propAddr = tenancies[tenancyIndex].property;
    address _tenant = tenancies[tenancyIndex].tenant;
    uint _earlyTerminationFee =
      tenancies[tenancyIndex].earlyTerminationFee;
    Property prop = Property(propAddr);
    require(msg.sender == _tenant || prop.owners(msg.sender) ==
      true);
    if (msg.sender == _tenant && newExpireTime <
      tenancies[tenancyIndex].expireTime) {
      paymentpool[tenancyIndex] += _earlyTerminationFee;
    }
    tenancies[tenancyIndex].expireTime = newExpireTime;
}
```

This function first gets the address of the property of the tenancy and the address of the tenant. Then it gets the value of the early termination fee on that tenancy. It then connects to the property contract using the address and requires that the current user is either the tenant of the tenancy or an owner of the property. Then, if the user is the tenant and the new expiration time is earlier than the old expiration time, it applies the early termination fee. Lastly, it changes the value of the expiration time for that tenancy. The fifth function, linkTenancyInfo() allows the user to update the info hash on a tenancy.

The last piece of the tenant registry contract was to create a payment system. We did this in almost the same way as it was done in the property contract, except in this case the user would need to specify the index of the tenancy they were making a payment on and payments could only go to an owner of the property of that tenancy. It would be great to support other kinds of payments, but due to scope constraints the main focus of this payment system was on making rent payments. First, we created the basic payable fallback function and `getBalance()` function. Then we created a function to get the balance of a specific tenancy by providing the tenancy index as input. From here, we coded the `sendPayment()` function.

```
function sendPayment(uint tenancyIndex, uint _amount)
public
payable
returns (uint)
{
   require(_amount <= msg.sender.balance);
   uint total = _amount +
     tenancies[tenancyIndex].currentBalanceForInterval;
   if (total >= tenancies[tenancyIndex].pricePerInterval) {
            tenancies[tenancyIndex].numPriceIntervals -= 1;
            tenancies[tenancyIndex].currentBalanceForInterval =
              total - tenancies[tenancyIndex].pricePerInterval;
            paymentpool[tenancyIndex] += _amount;
            return tenancies[tenancyIndex].keyCode;
   } else {
            tenancies[tenancyIndex].currentBalanceForInterval +=
              _amount;
            paymentpool[tenancyIndex] += _amount;
        }
    }
```

This is a payable function that takes a tenancy index and an amount as input. It then checks that the sender has enough funds in their account. Then, it calculates the current total for the current interval by adding the input amount and the current balance for the interval. If the total is greater than or equal to the price per interval, this means that the interval is paid and the number of intervals left on the tenancy should be decremented. Then the current balance for the new interval is set to be the difference between the total and the price per

interval in case the user overpaid. Then the amount is deposited into the payment pool and the keycode is returned because this means that at least one interval has been paid off by the tenant. If the total is not enough to pay off the interval, then the current balance for the interval is incremented by the input amount and the input amount is deposited in the payment pool. Lastly, we programmed the `receivePayment()` function.

```
function receivePayment(uint _tenancyIndex)
external
{
    address _propAddr = tenancies[_tenancyIndex].property;
    Property prop = Property(_propAddr);
    bool isOwner = prop.owners(msg.sender);
    require(isOwner == true);
    uint pmnt = paymentpool[_tenancyIndex];
    msg.sender.transfer(pmnt);
    paymentpool[_tenancyIndex] = 0;
}
```

This external function takes in the tenancy index of the tenancy the user would like to receive payments for. It then grabs the property address for that tenancy and connects to the property contract. From here it makes sure that the user is an owner of the property that the tenancy is in. Then it gets the amount of ETH for that tenancy in the payment pool and transfers it to the user. It then resets the payment pool value for that tenancy. With this contract done, we completed the back end development.

4.3 User Interface

After significant development work on the smart contracts in Ethereum and their integration with the truffle project, we ran into a large challenge. The truffle project was set up such that all of the UI work had to be done in HTML. While it is technically possible to do almost any UI development in HTML, it is incredibly time consuming and long code. After speaking to my supervisor, it was decided that it would be best to set up an additional react project for the front end work. React is an optimized JavaScript library for building UIs ("React"). It allows the developer to embed pieces of code in the JavaScript to make the

program more succinct and development much quicker. However, learning React is like learning an entirely new programming language, it is no small feat.

To do this, we began by finding a Codecademy course on React (Codeacademy). The course began by teaching the basic syntax of React. This involved nesting Javascript code, multiline code, nesting HTML, and ReactDOM. After this introduction, the course went on to cover building components, making the components interact, lifecycle methods, hooks, component states, and advanced React syntax. Each chapter of this course involved coding complex examples in React to ensure that the topics were being learned and took significant time. However, this was a necessary skill set to build in order to be able to build out the UI of our DApp.

From here, we looked for resources on how to build a DApp with React. Due to the topic being on the cutting edge of current development and research, there was only one detailed tutorial on how to get started (McCubbin, "How to Build Ethereum DApp with React JS"). Following this tutorial, the goal was to create a React app that would directly interact with the smart contracts we deployed on Ganache using a truffle project in the previous section. Successfully creating this interaction would be a source of significant time with essentially no resources on how to debug. However, the first piece of creating this React app was to make sure that node was installed and up to date. From here, we navigated to the correct directory and ran `npm install create-react-app` from the command line. This installed the basic setup for creating an app.

Next, we needed to create our app by running `create-react-app eth-todo-list-react`. As with any project, the next step was to install dependencies with `npm install`. This took significant time to debug due to version problems and incompatibilities within dependencies. Eventually, all of the correct dependencies were added. To verify that the app was initialized correctly, the next step was to launch the app using `npm run start`. This automatically opened up a chrome window with a blank UI using localhost - everything was set up correctly. The last step before starting development was just to delete the default style sheets.

Now, the real development work began. The parent component (or main component) for the app is the App.js file. So we began by trying to establish a connection to the local blockchain that was running with Ganache.

```
class App extends Component {
  componentWillMount() {
    this.loadBlockchainData()
  }

  async loadBlockchainData() {
    const web3 = new Web3(Web3.givenProvider ||
"http://localhost:7545")
    this.setState({web3})
    const accounts = await web3.eth.getAccounts()
    this.setState({ account: accounts[0] })
```

In this code, we established that the App is a component and that upon loading it, it should call the `loadBlockchainData()` function. This function connects with the Ganache blockchain running on local host 7545 and grabs the number of the account being used currently. The localhost number was originally set incorrectly, so this was fixed before the connection was successfully made.

Next came the connection to the deployed contracts. To do this, we first needed to go into the config.js file and add the contract addresses and ABIs as variables. A smart contract's ABI is JSON code that describes the structure of the contract including its functions. The DApp needed this to understand how it is supposed to communicate with the contract. These pieces of information were saved as variables and exported to App.js. Now, the property registry and tenant registry contracts were loaded into App.js.

```
const propertyRegistry = new web3.eth.Contract(PROPERTY_REGISTRY_ABI,
PROPERTY_REGISTRY_ADDRESS)
    this.setState({ propertyRegistry })
    const TENANT_REGISTRY_ADDRESS = await
propertyRegistry.methods.tenantRegistry().call()
    const tenantRegistry = new web3.eth.Contract(TENANT_REGISTRY_ABI,
TENANT_REGISTRY_ADDRESS)
    this.setState({ tenantRegistry })
```

One important piece to note is that we did add the property contract's ABI, but we didn't yet have any property addresses because those are added later using a function inside the property registry contract. We will return to how we handle this later. The code above loaded in the property registry contract that we launched on the local Ganache blockchain using the web3 connection to Ganache and the blockchain address of the contract. Since deploying the property registry also automatically deployed a tenant registry, we also loaded that contract in using its address. These were saved in the state of the component to be accessed later. This was the initial setup of the DApps connection to the contracts. From here we could start to pull information from them.

In order to pull information from the contract that does not require a transaction (i.e., no gas required), we simply loaded the variable value from the contract. This was done using the `call()` function which was built to interact with contracts in a way that may or may not involve gas. For example, we grabbed the owner value from the property registry contract using `const owner = await propertyRegistry.methods.owner().call()` and then set the state value using `this.setState({ owner })`. However, when trying to use functions in the contract to get some information, a different syntax was required.

The first attempt at calling the `getNumTenancies()` function of the tenant registry contract looked like `const numTens = (await tenantRegistry.methods.getNumTenancies().call()`. This resulted in the uninformative `Error: VM Exception while processing transaction`. As we would find out in the process of developing this React App, this is a common and frustrating error when trying to connect with smart contracts from the front end. In fact, this seemingly small bug was one of the largest challenges faced during front end development.

The debugging began by going back to remix and testing out the `getNumTenancies()` function in the contract. In remix, the function call worked perfectly. We also knew that the front end was correctly connected to the contracts. This meant that the error was in the creation of the function call, but there was no information as to how to fix it. After extensively searching the web, nothing seemed to work and this bug halted development for several days. Finally, in a Stack Overflow page, there was a `call()` example that had inputs for `from` as well as `gas` (Stack Overflow). This was the answer - the function call needed to include a gas limit and an account number from which the transaction was being sent. The resulting call was `const numTens = (await tenantRegistry.methods.getNumTenancies().call({ gas: "1000000", from: this.state.account }))`. This worked and pulled the information.

Now that we knew how to pull variables and call functions, we pulled in the owners of the registries, the number of properties and the number of tenancies. We then used the number of tenancies to pull the actual tenancies along with their information using a for loop.

```
const tenancies = []
    for (var j=0; j < numTens; j++) {
      const tenancyVals = await
tenantRegistry.methods.getTenancyAt(j).call({ gas: "2000000",
from: this.state.account })
      tenancies.push(tenancyVals)
    }
    this.setState({tenancies})
```

Then, the constructor initialized the state values and all that was left in the App component was the render function. This render function set the title of the page to be Property Properly App. It then loaded the property registry component followed by the tenant registry component. Based on what information from the App component was needed in these components, props were passed in. This included the tenancies that were loaded in the for loop, the number of tenancies, the number of properties, and etc. In the case of the property registry specifically, we passed in the property contract ABI so that the property registry component would be able to connect with properties that were created when the `addProperty()` function in the property registry contract was triggered by the front end.

The property registry component went through a significant amount of trial and error. Focusing on the render function, the first thing we did was to create the background box, a title, print the owner, print the number of properties, and then create a submission form for adding a property. This submission form had text boxes for the three inputs required for the `addProperty()` function in the contract: alias, geohash, and IPFS hash. Each of these input boxes had a corresponding on-change function in the component that recorded what was typed in. From here, the submit button triggered a `handleButtonClicked()` function which called the `addProperty()` function in the contract. Once the contract was deployed to the blockchain, the address was saved inside the property registry component.

The last piece of the render function was a list of the properties. We went through many structural iterations to do this and settled on creating a list of property components.

```
<p>Properties: </p>
<ul id="propertyList" className="list-unstyled">
        { this.props.keys.map((property) => {
            return(
               <div>
                  <Property key='{property}' addr={property}
                   web3={this.props.web3}
                   propertyABI={this.props.propertyABI}
                   account={this.props.account}/>
               </div>
            );
         })}
</ul>
```

This code iterated through the property addresses for any of the properties that have been added. For each of these, it then rendered a property component and passed in a few props to that component. From here, we created the property component which loaded in the connection to the property contract using the address and the ABI. It then rendered the alias, geohash, IPFS hash, and a boolean representing whether the user is an owner of that property. These were the basic pieces of information that we wanted the UI to display, so due to time and scope constraints we moved on to the tenant registry component. The goal was to add all

of the property registry contract's functions to the UI, but this became infeasible to the scope of the project. The focus of this DApp was really the complexity of the smart contracts, not the UI.

The last piece of the UI was the tenant registry component. The structure of this was very similar to that of the property registry component in that it rendered a title, the owner of the registry, the number of tenancies, and a submission form for adding a tenancy. This submission form required 3x the number of inputs as the `addProperty()` function. The same `onChange()` and `handleButtonClicked()` structure was used and the `handleButtonClicked()` function called the `addTenancy()` function in the deployed tenant registry contract. The final thing to render was a list of tenancies. We chose not to represent tenancies as their own components to remain synchronous with the back end structure and because we were able to access a list of the tenancies (and all of their information) from the tenant registry component. Therefore, we rendered the tenant address, property contract address for that tenancy, status, start time, expiration time, and tenancy info IPFS hash.

Property Properly App

Property Registry

Owner: 0x5060Eb538FaBD099dFdb345f3d29CFA366549395

Number of properties: 3

Add property:

Add alias... | Add geohash... | Add IPFS hash... | Submit

Properties:

Alias: Eiffel Tower

Geohash: u09tu

IPFS Metadata Hash: QmdcTYGrRj5Aw2UZxiDGMoGtVPiUwskw5vNNCWJKz9FR5e

Are you an owner? true

Alias: 123 Example Street

Geohash: gbsuv

IPFS Metadata Hash: QmZKw9nf5J8nVyHByUthVzcX8ZSxegJFLAoQDv8KnPfDxq

Are you an owner? true

Alias: test

Geohash: test

IPFS Metadata Hash: test

Are you an owner? true

Your account: 0x5060Eb538FaBD099dFdb345f3d29CFA366549395

Tenant Registry

Owner: 0x5060Eb538FaBD099dFdb345f3d29CFA366549395

Number of tenancies: 1

Add tenancy:

Add tenant address... | Add property address... | Add apartment number | Add start time... | Add end time...
Add early termination fe | Add keycode... | Add price per interval... | Add number of paymen | Add current value for in | Submit

Tenancies:

Tenant: 0x5060Eb538FaBD099dFdb345f3d29CFA366549395

Property: 0xba7091C6A98B08329E5D2670d687827b5D42dD01

Status: 0

Start time: 10121

End time: 123121

Tenancy info:

Your account: 0x5060Eb538FaBD099dFdb345f3d29CFA366549395

Figure 7: DApp User Interface

Deciding how all of these components should interact and which elements should be their own components took a long series of edits. We initially wanted more functions to be displayed, but we chose to prioritize the back end development instead. It was also challenging to figure out the best structure for which components to load in which pieces of

data. We initially tried loading the property contracts within the property registry component, but the properties themselves had far too much logic for this to work. So, the property contract ABI and address were passed as props to a separate property component that handled this logic. There were many structural decisions like this that made development quite challenging.

Chapter 5

Evaluation

5.1 Security Considerations

The blockchain is a very secure and efficient implementation of a digital distributed ledger. All transactions are available to every member computer (or node) on the network as they all contain a full copy of the blockchain. However, it is also possible for anyone to view transactions without being part of the network - they simply can't add to it. With the Ethereum blockchain, anyone with an internet connection can search for transactions using etherscan.io (Etherscan). This is certainly a privacy concern for those whose identity is connected to their cryptocurrency wallet because the details of the wallet are recorded in the transaction. In addition, it is very difficult to anonymously purchase any cryptocurrency with fiat currency (e.g.., dollars, euros, etc.) because most cryptocurrency exchanges are governmentally mandated to record the purchaser's identity (Peters).

This brought us to another key security concern of working with the blockchain - it can and has been used to commit crimes and harm people. An online marketplace called the Silk Road was created on the Bitcoin blockchain and was used to sell illegal drugs, weapons, and even assassins for hire (Christin). Without proper care and regulation, a blockchain based application could be used to cause harm. This is something that has been directly considered when developing this DApp to avoid harm to users.

Since my DApp is on the Ethereum blockchain, we needed to consider the security vulnerabilities of the Ethereum blockchain specifically. In the DAO attack of 2016, a hacker took advantage of a reentrancy vulnerability to steal $60 million (Meier). Since Ethereum handles the majority of blockchain traffic, it also has a large attack surface. One paper cited 40 different security vulnerabilities in the Ethereum blockchain - we can be sure there are many more we don't know about (Chen). While there isn't space to discuss all 40 vulnerabilities, we will go into the details of a few major ones.

One of the largest vulnerabilities is due to the decentralized nature of the Ethereum blockchain. If a hacker were able to control over 50% of the block verification work, they could create their own main chain - effectively controlling the blockchain (Chen). Another vulnerability involves a DoS (Denial of Service) attack in which a hacker provides such a high reward for their transactions to be put on the chain that they can temporarily block other transactions from being processed (Chen). This works because the order in which transactions are added to the blockchain is determined by which transactions are chosen by miners to be constructed into a block. A miner's incentive is to pick the transactions with the highest gas price (eg. reward).

Another very important vulnerability within the Ethereum blockchain is due to the type system of the language in which Ethereum smart contracts are written - Solidity (Chen). It is possible to call a different smart contract within a function of your smart contract and Solidity is unable to prevent this. If the contract being called is malicious, this contract's code can be run through a perfectly harmless looking mask contract (Chen). In addition to this, Solidity is also known to have a buggy compiler that results in even more vulnerabilities. Our DApp is also written in Solidity, therefore the security vulnerabilities of the language are security vulnerabilities of my project.

5.2 Solidity Testing

We also created some basic tests inside of remix. These included unit testing and assertions. This kind of testing is helpful in making sure that all functions are running as expected. One basic test example was checking that the `msg.sender` account was equal to the tester account we expected to be using and that the value of that account was what we expected (Remix).

```
function checkSenderAndValue() public payable {
    Assert.equal(msg.sender, TestsAccounts.getAccount(1),
      "Invalid sender");
    Assert.equal(msg.value, 100, "Invalid value");
}
```

(Remix)

5.3 Strengths

The Property Properly DApp has many strengths and there were pieces of the development that went very well. The first piece to acknowledge is that this project required learning and implementing code from scratch in two programming languages that we have no previous experience with. The Solidity programming language, which is used for writing smart contracts, is completely unique from other languages ("Solidity Programming Language"). There are resources for learning this language, but the majority of the skill building came from trial and error with our own contracts. We were able to learn Solidity well enough to create and deploy interconnected smart contracts from scratch. We also learned how to use React by doing a multi-day course on Codecademy (Codeacademy). React, in particular, is not known to be an easy language to develop with. However, we were able to learn React well enough to code a React project from scratch.

Although there were some limitations to the UI, we still developed a DApp with a functional and straightforward UI in React. This UI successfully interacted with the contracts, displayed information, and deployed contracts. This is much less straightforward to develop than a regular app with a server or database on the back end. It should also be noted that our particular combination of functionalities has never been done before in any open source DApp.

Our DApp created a working property registry and tenant registry that could be used for real properties. In addition, we supported practical functionalities like having multiple owners on a property and being able to link external files in the form of IPFS hashes. Our DApp also contained a functional payment system for both properties and tenancies to make payments on properties to a third party and to make rent payments on tenancies to the property owner. We can even support short-term rental tenancies because the owner selects

the number of payments and amount per payment upon completion of a tenancy. Then, when the first payment is made in full, the DApp can release a door keycode to the user. We also created the functionality for a user to move up the expiration time on their tenancy with the early termination fee automatically applied. All of these details make the DApp uniquely suited to real-world applications.

5.4 Limitations

Along with the strengths, there are also some limitations of our DApp due to scope and time constraints. These limitations would be excellent areas to concentrate future work on the project. While we were able to learn Solidity and React from scratch, the lack of development experience in these languages means that there were certain limitations in programming. For example, we are still learning and understanding how the system of lifecycle functions work in React. This means that there may be pieces of code inside of a certain lifecycle function that would be better suited to a different one.

In addition, given more time, it would be ideal to expand the scope of the UI to encompass all of the functionality of the back end. There are a few functions within the property contract and the tenant registry contract that could use submission forms in the UI that trigger the functions in the back end and reflect that change. It would also be ideal to be able to decode and render some of the hashes that contain image files or documents. We attempted multiple strategies to render these to the front end without luck. This is certainly something that we improve the presentation of the DApp.

It should also be noted that there is a significant expense in deploying the contract to the blockchain. Luckily, this only needs to be done once. There is also an expense to every transaction that occurs within the contracts. This is something to be mindful of as the congestion of the network and therefore the gas price continues to increase.

Finally, one area for future work is in the complexity of the payment system. It would be ideal to support non-rent payments on a tenancy and generally increase the complexity of the payments that can be handled by the contracts.

Chapter 6

Conclusions & Future Work

6.1 Conclusion

We conclude that it is possible to utilize the unique features of the blockchain to develop a property ledger and transaction management system. We find that the best approach to accomplishing this task is to create a decentralized application on the Ethereum blockchain. Within this application, we deployed contracts for a property registry, tenant registry, and each unique property within our registry.

One limitation of using the Ethereum blockchain is that there is an expense involved in every transaction. However, we determined that the expense involved was comparable to, if not less than, the expenses that a bank would charge to transfer funds for example. In addition, the cost of lawyers and other middlemen without our system can be far more than Ethereum transaction costs (Law Guideline).

Additionally, we find that the best way to store metadata is to put files and images into the IPFS and generate unique IPFS hashes for them. We would then store those hashes as a variable within each property or tenancy. Some interactions with our contracts were designed not to incur any costs because they don't involve creating a transaction on the blockchain. This, for example, included getting the number of properties or getting the IPFS hash for a property. Other interactions with the contracts necessitated a transaction on the blockchain such as adding a property or adding an owner to a property.

We were also able to set up a payment system to allow owners to make payments on a property to an outside party and tenants to make payments to the property owner. We were even able to add a functionality that would automatically release the door keycode to a tenant once they made their first payment. This worked by storing funds in the contract until they were transferred to the specified party when requested. Since the contracts were deployed on

the blockchain, this means that all of these payments were permanently recorded on that blockchain.

In this way, we were able to utilize the immutability and trustlessness of the blockchain to create a more effective, reliable property ledger and transaction management system.

6.2 Future Work

There are several directions that future work could take from this point. Firstly, the DApp developed within this dissertation could be expanded to handle more complex transactions and payments. Our project can handle mortgage payments on properties, but it would be very useful to set up an automated way to handle escrow payments such that the deed to the house is automatically signed and forwarded to the buyer once the seller accepts the offer and payment is taken out of escrow. We know this can be done because our DApp already sends the keycode of the door of a property to the tenant once the first rent payment is made.

Secondly, a similar structure could be used to manage other property assets. It could be useful to separate the land and the building(s) on that land into separate components to allow for a separation of assets. For example, it is possible for the land and the building to be owned by separate entities. It is also possible for an entity to have certain rights on the land such as water or oil rights. It could be an interesting project to explore the creation of a rights registry for pieces of land. The ownership of these rights could be tracked in the blockchain. The rights could be licensed out for a fee or sold through smart contracts.

Thirdly, it would be interesting to explore blockchain applications to other areas of real estate such as representing properties as securities and creating a marketplace to trade them. In fact, this has already been successfully piloted by the St. Regis Resort in Aspen. AspenCoin raised $18 million dollars upon initial investment (Aspen Digital Inc.). This strategy of tokenizing property securities could be more widely used in commercial real estate. One very interesting project could be to create an application that would allow

companies to tokenize large commercial properties in this way. This would create a lucrative, alternative way for companies to raise capital. In addition, this could open up real estate investing to the average person by allowing investment through cryptocurrency.

Bibliography

"An Unbiased Global Financial System." MakerDAO, makerdao.com/en/.

Aspen Digital Inc. Aspen Coin, 2018, www.aspencoin.io/.

"Bitcoin." Overstock, help.overstock.com/help/s/article/Bitcoin.

Buterin, Vitalik. "Eip-20: Token Standard." Ethereum Improvement Proposals, 19 Nov. 2015, eips.ethereum.org/EIPS/eip-20.

Buterin, Vitalik. "Ethereum Whitepaper." Ethereum.org, 2013, ethereum.org/en/whitepaper/.

Chen, Huashan. "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses." ACM Digital Library, 2021, dl.acm.org/doi/fullHtml/10.1145/3391195#sec-19.

Chohan, Usman W. "The Problems of Cryptocurrency Thefts and Exchange Shutdowns." SSRN, 7 Mar. 2018, papers.ssrn.com/sol3/papers.cfm?abstract_id=3131702.

Chou, Jonathan. "The Bee Token: Decentralized Short-Term Housing Rentals." The Bee Token | Decentralized Short-Term Housing Rentals, 2017, www.thebeetoken.com/.

Christin, Nicolas. "Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace." Proceedings of the 22nd international conference on World Wide Web. 2013.

"Coinbase: Buy & Sell Bitcoin, Ethereum, and More with Trust." Coinbase, www.coinbase.com/.

ConsenSysMesh. "Consensysmesh/Real-Estate-Standards." GitHub, Consensys, github.com/ConsenSysMesh/real-estate-standards.

CryptoKitties. "Collect and Breed Digital Cats!" CryptoKitties, www.cryptokitties.co/.

Davis, Aaron. MetaMask, metamask.io/.

Dexaran. "Erc223 Token Standard Reference Implementation." GitHub, github.com/Dexaran/ERC223-token-standard.

Domingo, Carlos. "Securitize.io." Securitize, 2017, securitize.io/.

Entriken, William, et al. "Eip-721: Non-Fungible Token Standard." Ethereum Improvement Proposals, 24 Jan. 2018, eips.ethereum.org/EIPS/eip-721.

"Etherscan: The Ethereum Blockchain Explorer." Etherscan, 2021, etherscan.io.

Glickman, Edward. Introduction to Real Estate Finance. Elsevier Academic Press, 2020.

"Gods Unchained ." Gods Unchained, 2021, godsunchained.com/.

Goitom, Hanibal. "Our New Reports on Regulation of Cryptocurrency around the World." In Custodia Legis: Law Librarians of Congress, 13 July 2018,

blogs.loc.gov/law/2018/07/our-new-reports-on-regulation-of-cryptocurrency-around-the-world/.

Gupta, Sourav Sen. “Blockchain: The Foundation Behind Bitcoin.” Indian Statistical Institute, 2021, www.isical.ac.in/~debrup/slides/Bitcoin.pdf.

“Introduction to Dapps.” Ethereum.org, 2 Sept. 2021, ethereum.org/en/developers/docs/dapps/.

“Introduction to Smart Contracts.” Ethereum.org, 6 Aug. 2021, ethereum.org/en/developers/docs/smart-contracts/.

Karayaneva, Natalia. “Real Estate Transaction Automated.” Propy, 2021, propy.com/browse/.

Kirby, Carrie. “Person to Person Bitcoin Exchanges Spread across the Globe.” CoinDesk, 2013, www.coindesk.com/markets/2013/10/10/in-person-bitcoin-exchanges-spread-across-the-globe/.

Lewenberg, Yoad, et al. "Bitcoin mining pools: A cooperative game theoretic analysis." Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems. 2015.

“Limited Liability.” Legal Information Institute, Legal Information Institute, www.law.cornell.edu/wex/limited_liability.

Lipton, Alex, and Stuart Levi. “An Introduction to Smart Contracts and Their Potential and Inherent Limitations.” The Harvard Law School Forum on Corporate Governance, 26 May 2018, corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations/.

Marr, Bernard. “A Very Brief History of Blockchain Technology Everyone Should Read.” Forbes, Forbes Magazine, 20 Mar. 2018, www.forbes.com/sites/bernardmarr/2018/02/16/a-very-brief-history-of-blockchain-technology-everyone-should-read/?sh=1f1fe6b97bc4.

Marshall, Rob, director. Mary Poppins Returns. Walt Disney Pictures, 2018, www.disneyplus.com.

McCall, Matt. “A $200 Million Pizza! Here's How Bitcoin Made That Possible ...” Nasdaq, www.nasdaq.com/articles/a-%24200-million-pizza-heres-how-bitcoin-made-that-possible-...-2020-12-02.

McCubbin, Gregory. “How to Build Ethereum Dapp with REACT.JS · Complete Step-by-Step Guide.” Dapp University, 2021, www.dappuniversity.com/articles/ethereum-dapp-react-tutorial.

McCubbin, Gregory. “The Ultimate Ethereum Dapp TUTORIAL (How to Build a Full Stack Decentralized Application Step-By-Step).” Dapp University, 3 Aug. 2021, www.dappuniversity.com/articles/the-ultimate-ethereum-dapp-tutorial.

McCulloch, Samuel. RealT, 2021, blog.realt.co/.

Meier, Julia, and Benedikt Schuppli. "The DAO Hack and the Living Law of Blockchain." Digitalisierung–Gesellschaft–Recht: Analysen und Perspektiven von Assistierenden des Rechtswissenschaftlichen Instituts der Universität Zürich (2019): 27-43.

Mi, Remco. “Ethereum Accounts.” Ethereum.org, 5 Sept. 2021, ethereum.org/en/developers/docs/accounts/.

Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." Decentralized Business Review (2008): 21260.

Narayanan, Arvind. “Bitcoin and Cryptocurrency Technologies : A Comprehensive Introduction.” Princeton University, The Trustees of Princeton University, press.princeton.edu/books/hardcover/9780691171692/bitcoin-and-cryptocurrency-technologies.

“Oracles.” Ethereum.org, 27 Aug. 2021, ethereum.org/en/developers/docs/oracles/.

Peters, Gareth, Efstathios Panayi, and Ariane Chapelle. "Trends in cryptocurrencies and blockchain technologies: A monetary theory and regulation perspective." Journal of Financial Perspectives 3.3 (2015).

Quarshie, Perrin. “Building a Better Alternative .” RealBlocks, 2021, www.realblocks.com/home.

Radomski, Witek, and Andrew Cooke. “Eip-1155: Multi Token Standard.” Ethereum Improvement Proposals, 17 June 2018, eips.ethereum.org/EIPS/eip-1155.

“React – a JavaScript Library for Building User Interfaces.” – A JavaScript Library for Building User Interfaces, reactjs.org/.

“ReactJS Tutorial Part I: LEARN Reactjs for Free.” Codecademy, Codecademy, www.codecademy.com/learn/react-101.

“Remix Ethereum IDE.” Remix, remix.ethereum.org/.

Richardson, Brenda. “Housing Market Gains More Value in 2020 than in Any Year since 2005.” Forbes, Forbes Magazine, 29 June 2021, www.forbes.com/sites/brendarichardson/2021/01/26/housing-market-gains-more-value-in-2020-than-in-any-year-since-2005/?sh=202fdb214fe0.

Smith, Julie, et al. “Tokenized Securities & Commercial Real Estate.” MIT Management Sloan School, Digital Currency Initiative - MIT Medial Lab, 14 Mar. 2019, mitcre.mit.edu/wp-content/uploads/2019/11/Tokenized-Security-Commercial-Real-Estate2.pdf.

“Solidity by Example.” Solidity by Example | 0.8.3, solidity-by-example.org/.

“Solidity Programming Language.” Solidity Programming Language, soliditylang.org/.

Stack Overflow. “Calling Solidity Functions to Reactjs.” Stack Overflow, 2019, stackoverflow.com/questions/55195303/calling-solidity-functions-to-reactjs.

Suum Cuique Labs GmbH. Hashmasks, 2021, www.thehashmasks.com/.

Szabo, Nick. “Bit Gold.” Bit Gold , 29 Dec. 2005, nakamotoinstitute.org/bit-gold/.

Taylor, Michael Bedford. "The evolution of bitcoin hardware." Computer 50.9 (2017): 58-66.

Tricchinelli, Rob. “Bitcoin Deemed 'Money' under D.c. Financial Services Law .” Bloomberg Law, 2020, news.bloomberglaw.com/banking-law/bitcoin-deemed-money-under-d-c-financial-services-law.

“Uniswap.” Uniswap Blog RSS, uniswap.org/blog/uni/.

“Virtual Currency.” Department of Financial Services, www.dfs.ny.gov/apps_and_licensing/virtual_currency_businesses/bitlicense_faqs.

Wackerow, Paul. “Gas and Fees.” Ethereum.org, ethereum.org/en/developers/docs/gas/.

Wang, Wenbo, et al. "A survey on consensus mechanisms and mining strategy management in blockchain networks." IEEE Access 7 (2019): 22328-22370.

“What Is the Average Cost of a Real Estate Attorney.” Law Guideline, 5 Nov. 2019, lawguideline.org/what-is-the-average-cost-of-a-real-estate-attorney/.

Wosnack, Nathan. “One Block at A Time®.” Ubitquity, 2021, www.ubitquity.io/.

Yilmaz, Ensar. “Erc-20 Token Standard.” Ethereum.org, 2021, ethereum.org/en/developers/docs/standards/tokens/erc-20/.

Zheng, Zibin, et al. "An overview of blockchain technology: Architecture, consensus, and future trends." 2017 IEEE international congress on big data (BigData congress). IEEE, 2017.

Zheng, Zibin, et al. "Blockchain challenges and opportunities: A survey." International Journal of Web and Grid Services 14.4 (2018): 352-375. 2018.

Zheng, Zibin, et al. “An Overview on Smart Contracts: Challenges, Advances and Platforms.” Future Generation Computer Systems, North-Holland, 17 Dec. 2019, www.sciencedirect.com/science/article/abs/pii/S0167739X19316280?casa_token=JnAxV1MvwYUAAAAA%3ArpXO8LEeb7XPwvbhsX8aKQUTAsn7jRT3VTq3Ms-twu0sQUPhbEWWfo7A1BfIQOls5JFaJi8.