UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Blockchains and the evolution of smart contract languages: an overview

Sil Vaes

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

2022
2023

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Blockchains and the evolution of smart contract languages: an overview

Sil Vaes

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Jan VAN DEN BUSSCHE

Abstract

Blockchain is a rapidly moving and complex field. Due to the recent buzz surrounding blockchain systems it is difficult to find objective information. This thesis provides an overview of three blockchain systems and the evolution of the technology behind them, with a particular emphasis on the languages used to program on the blockchain.

The three blockchain systems presented in this thesis are Bitcoin, Ethereum, and Cardano.

Bitcoin is the pioneer of blockchain technology and the most well known application. Bitcoin's programming language, Script, is intentionally limited in its capability but was the first to show that programmable transactions are possible.

The next blockchain, Ethereum, often described as a "world computer", is a general-purpose blockchain. Ethereum is fully programmable and the most common programming language is Solidity. In contrast to Script, Solidity is a high-level language to program more complex smart contracts for the Ethereum blockchain. This versatility does come with disadvantages however, being able to express complex programs also means that security becomes an issue.

The final blockchain presented is Cardano. Cardano is a research-first blockchain system and tries to find the middle ground between Bitcoin and Ethereum. Where Bitcoin is very secure due its limited programmability and Ethereum provides a sophisticated programming environment, Cardano attempts to combine the best of both worlds. The programming languages discussed here are Plutus and Aiken. Both of these languages are purely functional languages which makes is much easier to verify smart contracts before deploying them, which increases security.

The evolution mirrors the general trend in software development, but in a shorter timeframe. The struggle to balance security, ease-of-use, and expressiveness can be found too. Bitcoin started with a simple language, akin to the early days of programming. Then Ethereum shifted toward a more complex, but also more powerful language, which mirrors the rise of languages like C++ and Java. Finally, Cardano seems to embody a contemporary mindset which adopts more functional language features, which again mirrors the rise of languages like Rust.

Abstract (Dutch)

Blockchain is een snel evoluerend en complex studiegbied. Door de recente buzz rondom blockchain systemen is het moeilijk om objectieve informatie te onderscheiden. Deze thesis geeft een overzicht van drie blockchain systemen en de evolutie van de technologie erachter, met een bijzondere nadruk op de talen die gebruikt worden om op de blockchain te programmeren.

De drie blockchain-systemen die in deze thesis worden gepresenteerd zijn Bitcoin, Ethereum en Cardano.

Bitcoin is de pionier van de blockchaintechnologie en de meest bekende toepassing. De programmeertaal van Bitcoin, Script, heeft opzettelijk beperkte mogelijkheden, maar toonde als eerste aan dat programmeerbare transacties mogelijk zijn.

De volgende blockchain, Ethereum, vaak beschreven als een "wereldcomputer", is een blockchain voor algemene doeleinden. Ethereum is volledig programmeerbaar en de meest gebruikte programmeertaal is Solidity. In tegenstelling tot Script is Solidity een high-level taal om complexere smart contracts voor de Ethereum blockchain te programmeren. Deze veelzijdigheid heeft echter ook nadelen, het kunnen uitdrukken van complexe programma's betekent ook dat beveiliging een probleem is.

De laatste blockchain die wordt besproken is Cardano. Cardano is een research-first blockchainsysteem en probeert de gulden middenweg te vinden tussen Bitcoin en Ethereum. Waar Bitcoin erg veilig is door de beperkte programmeerbaarheid en Ethereum een geavanceerde programmeeromgeving biedt, probeert Cardano het beste van beide werelden te combineren. De programmeertalen die hier worden besproken zijn Plutus en Aiken. Beide talen zijn puur functionele talen die het veel gemakkelijker maken om smart contract te verifiëren voordat ze worden ingezet, wat de veiligheid verhoogt.

De evolutie weerspiegelt de algemene trend in de softwareontwikkeling, maar in een korter tijdsbestek. De strijd om een balans te vinden tussen veiligheid, gebruiksgemak en expressiviteit is hier ook terug te vinden. Bitcoin begon met een eenvoudige taal, verwant aan de begindagen van programmeren. Daarna verschoof Ethereum naar een complexere, maar ook krachtigere taal, die de opkomst van talen als C++ en Java weerspiegelt. Tot slot lijkt Cardano een eigentijdse mindset te belichamen die meer functionele taalkenmerken gebruikt, wat weer de opkomst van talen als Rust weerspiegelt.

Contents

Chapter 1

Introduction

Blockchain technology has recently emerged at the forefront of a technological revolution, promising profound societal impacts far beyond its initial conception. Regardless of whether one aligns with the wave of optimism or remains skeptical, it is becoming increasingly challenging to dismiss the relevance of blockchain systems within the contemporary world.

This thesis aims to provide an overview and understanding of blockchain systems, with a particular emphasis and the languages used to program smart contracts on the blockchain platform. An important question is whether these languages follow conventional software development wisdom or completely differ from general-purpose languages. Moreover, if these are different, how have these adapted to the needs of blockchain systems? Besides the smart contract programming languages, this thesis will also explore the evolution of different aspects of the blockchain system, such as consensus mechanisms and transaction models.

There has been considerable hype associating blockchain with the potential to address numerous global challenges. While the buzz surrounding blockchain technology is immense, it is essential to tread carefully when navigating the wealth of information. Unfortunately, this has led to many unreliable sources, biased articles, and even misinformation. This thesis tried to weed out the hyperbole to find reliable and factual information to examine current blockchain systems objectively.

Having a proper understanding of blockchain and its components is essential. Blockchain systems have suffered severely from various security breaches in their short history. How do these happen, and is there something that can be done? Or is there already something being done? These are essential questions, and these issues could be solved technologically.

The programming languages used to program sensitive applications on the blockchain play an essential role in securing and protecting people from malicious actors. Understanding the mechanisms of running the program itself is also crucial to write well-defined and safe programs. Knowing the underlying mechanisms is analogous to writing low-level programs. Programmers must understand how their programs will run on a particular piece of hardware; this is not any different from applications running on the blockchain. The environment is entirely different from traditional software development and requires different thinking. A programmer will have to keep in mind how his program will interact with a certain type of transaction or state of the blockchain. This thesis also attempts to provide the reader with the necessary skill set to tackle programming on the blockchain.

The thesis is structured into four main chapters, each building upon the prior, to

provide a comprehensive understanding of the evolution of blockchain systems.

The first chapter, prerequisites, lays out the groundwork for blockchain technology. This chapter is an example-driven introduction to blockchain systems using a land registry blockchain implementation to explain the basics for readers unfamiliar with its inner workings, creating a foundation to understand the subsequent sections.

This introduction is crucial to understand the rest of the thesis, which will go into more complex topics, such as how the different consensus mechanisms work and transaction models. Basic cryptography, decentralized systems, and consensus are explained in this chapter. Here a basic blockchain implementation programmed in Rust is also presented to show an example of how someone would program a rudimentary blockchain system.

The following chapter is dedicated to Bitcoin. Bitcoin is the pioneer of blockchain technology and the most well-known application. This chapter will detail the address scheme, transaction model, proof-of-work, and Script. Bitcoin's Script language, while non-Turing complete, was the first to demonstrate programmable transactions on a blockchain. This chapter also contains examples of the most commonly used Script programs.

The next chapter, Ethereum, will focus on Ethereum, a second-generation blockchain system. Ethereum is a general-purpose blockchain system, with its claim to fame being its programmability. Being programmable means, it has a much more sophisticated programming language called Solidity. The evolution from basic scripting in Bitcoin to Solidity in Ethereum will be explored in depth. The transaction model and consensus mechanism are also essential and will get the attention they deserve.

Following Ethereum, the chapter on Cardano will dive into the third generation of blockchain systems. Cardano differs from Bitcoin and Ethereum because it is a research-first blockchain project. Cardano looked back at Bitcoin and Ethereum and tried to find a middle ground. Where Bitcoin is very limited in programming capabilities, Ethereum goes to the other extreme and provides general-purpose programming with security implications. Cardano tries to find a balance between these two extremes. The programming languages discussed in this chapter are Plutus and Aiken. Both are purely functional languages which has the advantage of facilitating the verification of smart contracts.

Plutus is a notoriously difficult programming language, which is why this thesis will provide an alternative. Aiken is a fairly new programming language with a better developer experience and more familiar syntax compared to Plutus. It takes inspiration from modern programming language design like Rust and Typescript to deliver a more satisfying smart contract environment.

The final chapter contains the conclusion of the thesis. This chapter will provide an overview of the evolution of blockchain systems and smart contract languages. This chapter will also reflect on the master's thesis and what has been learned.

Throughout this thesis, the intricacies of blockchain technology and the complexities of smart contract languages are clarified, offering a systematic understanding of their evolution, capabilities, and potential. Through the chronological study of Bitcoin, Ethereum, and Cardano, this thesis provides a comprehensive overview of how smart contract languages have progressed, what problems they have solved, and what challenges remain.

Many sources have been used to research and understand different blockchain systems. These are listed here for ease of reading and consulting.

For the prerequisite chapter the following two resources:

- Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction [Nar+16].
- Blockchain Technology Overview [Yag+18].

The Bitcoin chapter used the following sources:

- Serious Cryptography: A Practical Introduction to Modern Encryption [Aum17].
- Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction [Nar+16].
- Mastering Bitcoin: Programming the Open Blockchain [Ant17].
- The Bitcoin Developer Guide [Dev23].

The chapter on Ethereum required the following to write:

- Mastering Ethereum: Building Smart Contracts and DApps [AW18]
- Ethereum: A secure decentralised generalised transaction ledger [Woo22].
- The Ethereum Development Documentation [Eth23].
- Ethereum Smart Contract Development in Solidity [Zhe+20].
- The Foundry Documentation [Tea23].
- Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond [Edg23].
- OpenZeppelin Documentation [BA23].

Finally, the Cardano chapter required the following:

- Cardano for the Masses: A financial operating system for people who don't have one [Gre22].
- Cardano Developer Portal [IOG23].
- Aiken: the Future of Smart Contracts [Ros23].
- An Introduction to Plutus Core [Gal21].
- Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol [Kia+17].
- Ouroboros-BFT: A Simple Byzantine Fault Tolerant Consensus Protocol [KR18].
- Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain [Dav+18].

Chapter 2

Prerequisites

This chapter is dedicated to a thorough examination of blockchain technology and will use land registration as example throughout this chapter. The motivation for this focus arises from the existing issues inherent to conventional land registration systems, which facilitates further exploration of alternative, more efficient methodologies.

Traditional land registration systems, despite their historical efficiency, are increasingly scrutinized for their various limitations. Commonly observed problems include human error, inaccuracies, susceptibility to manipulation and fraud, as well as prolonged and costly transactions [Int11]. Additionally, the processes of data access and verification present significant challenges due to outdated infrastructures and lack of centralized, real-time data access.

These issues underscore the need for comprehensive innovation within land registration systems. Rather than adopting a piecemeal approach to reform, this chapter suggests a restructuring of the system, one that integrates modern technologies to holistically address the identified challenges.

Blockchain technology is introduced as a potential answer. To ensure a comprehensive understanding, the chapter first explores decentralized systems, cryptography, and automated contracts as independent solutions. Each component's individual strengths and weaknesses are examined in order to understand their potential impact and limitations.

The discussion then proceeds to blockchain technology as a unified solution that amalgamates the aforementioned components. The chapter delves into the core elements of blockchain, including Distributed Ledger Technology (DLT), cryptography, consensus mechanisms, and smart contracts. Then a section shows how they interact to form a cohesive whole.

The subsequent section conceptualizes a land registration process underpinned by blockchain technology, and how it could serve to mitigate the issues prevalent in traditional systems. Each phase of the blockchain-based land registration process will be dissected to comprehend how blockchain addresses the identified challenges.

Upon completion of this chapter, readers should gain a solid understanding of blockchain technology.

As we move towards a more digital and interconnected world, the need for a more robust, secure, and efficient system has become evident.

2.1 Land Registration Challenges

Land registration systems record the ownership of land and property and are fundamental to societal functions, facilitating transactions and disputes resolution. However, traditional land registration systems are fraught with a series of challenges that often result in inefficiency, inaccuracies, and a lack of trust among stakeholders.

In traditional land registration systems, a significant amount of the process is handled manually, which makes it highly susceptible to human error. Mistakes in data entry, updates, and validation can lead to inaccuracies in the registry. In some cases, these inaccuracies may cause serious disputes over land ownership that are costly and time-consuming to resolve.

Land registration systems are frequently centralized, managed by a single entity or authority. This centralization often leads to a lack of transparency and accountability, providing opportunities for manipulation and fraud. False documentation and identity theft are among the types of fraudulent activities that can occur, resulting in unauthorized transfers and registrations.

The process of land registration often involves numerous steps, each requiring a considerable amount of time and resources. This includes multiple verification stages, document handling and approvals, which often span across different departments or authorities. Not only does this result in lengthy transaction times, but it also incurs significant administrative costs.

Access to information in traditional land registration systems is often impeded by bureaucratic red tape and outdated technologies. As a result, verifying the authenticity and accuracy of records is often a challenging and time-consuming task. Furthermore, the lack of real-time data updates can lead to instances where multiple parties unknowingly base decisions on outdated information, thus causing conflicts.

These challenges underscore the limitations of traditional land registration systems. This solution ideally would address these previously mentioned issues. The subsequent sections will explore technology-driven solutions to these issues.

2.2 Solutions

The previous section discussed issues with the current iteration of land registration systems. This section will attempt to address these and find possible technological solutions. These solutions include cryptography, decentralized systems, and automated contracts.

Cryptography

Using cryptography it is possible to create a land registration system. Imagine a land registration system with two rules.

The first rule is that a central authority, here called the “Registry”, can create and transfer land titles whenever it wants. To create a land title, it creates a new ID and constructs a string with this ID. It then computes the digital signature of this string with its private key. This string together with its signature, is a land registration token. Anyone can verify the validity of this token by checking if the signature is valid for the string.

Here follows a short explanation of digital signatures. A digital signature is supposed to be an analog to a handwritten signature. It thus has two properties. First, only the owner of the signature can create the signature, but anyone who can see the signature can verify it is valid. Secondly, the signatures has to be tied only one document as

to not allow the signature to be used to agree to a different document which was not signed by the signature holder at all.

The above discussion is very intuitive, below is a more concrete explanation of what is meant by it. A digital signature scheme consists of the following three algorithms:

- `generate_keys(keysize) →(sk, pk)`: the `generate_keys` function takes a key size and generates a key pair. The private key, `sk`, should never be given out to any other users and remains a secret to everyone. The private key is used to sign messages. The public key, `pk` is the verification key which can be shared publicly. Anyone with access to the public key can use it to verify signatures created with the corresponding private key. This function is randomized as to generate different keys for different people.
- `sign(sk, message) →sig`: the sign function takes a message and a private key as inputs and outputs a signature for the message using the private key.
- `verify(pk, message, sig) →bool`: the verify function takes a message, a signature, and a public key as input. It returns true if that signature is valid for a message using the public key, and false otherwise. This allows anyone to check if a message is signed by the owner of a public using a certain signature. This function as to be deterministic.

The following two properties have to hold under a digital signature scheme:

- Any valid signature must verify using the verify function.
- Signatures must be unforgeable. Which means that if someone has access to a public key and sees the corresponding signatures, he cannot forge the signature for arbitrary messages.

Another convenient property is that public keys can be equated to an identity of a user. If another user receives a message with a signatures which is verified with a public key `pk`, then the receiver can think of `pk` stating the message. Thus, it is possible to think a public as an actor in a system who can create messages by signing those messages. From this point-of-view, the public key is an identity and for someone to use this identity, he has to have access to the corresponding public key, `sk`.

The second rule is that whoever owns a token can transfer it to anyone else. This is not done by simply sending the token datastructure to the recipient, but by using cryptographic operations.

For example, the Registry wants to send a token to Alice. To do this, the Registry creates a statement which says "Give this to Alice", where this is a hash pointer which references the token. Note that identities are simply pubic keys, so Alice refers to one of her public keys in this case. Next the Registry signs the serialized representation of the statement. Since the Registry originally owned the token, it has to sign any statements which transfer the token. Once this statement is signed, Alice is the owner of the token, and thus owner of the property. Alice can prove to anyone that she owns it, because she can present the datastructure with the Registry's signature and it points to a valid token.

Now follows a short explanation of hash functions and hash pointers. A general hash function is a mathematical function with the following properties:

- Its input can be of an arbitrary size.
- Its output has a fixed size.
- It is deterministic.

- It is efficiently computable.

Cryptographic hash functions have three more properties: collision resistance, hiding, and avalanche effect.

Collision resistance means that it is very unlikely that two different inputs produce the same output. In practice this means that nobody can find a collision. Hashes serve as a summary of piece of data or message. This is an efficient way to "remember" what has already been seen. A big piece of data, multiple gigabytes, has a hash of 256 bits for example. This greatly reduces the required storage.

The hiding property says that given an output of a hash function, $H(x) = y$, there is no feasible way to determine the input, x, to produce the output. This obviously only holds if the input space is sufficiently large. If, for example, the input space contains only contains two elements, an attack can simply hash both elements and determine which of the two input elements produce which hash.

The avalanche effect says that small changes on the input have a large impact on the output of a hash function. Even a minor change in input should result in a significantly different hash, making it difficult to infer similarities between similar inputs.

Now a crucial datastructure using hashes will be discussed, namely hash pointers. A hash pointer is simply a pointer to data together with the cryptographic hash of that data. The additional cryptographic hash allows users to verify that the data has not been changed in addition to retrieving the data using the pointer. Hash pointers can be used in many datastructures. The most basic one is simply replacing existing pointers with hash pointers.

A linked list with its pointers replaced with hash pointers is one such example. This datastructure is called a blockchain. In a normal linked list each block contains data and a pointer to the previous block. In a blockchain each blocks not only contains the location of the previous data block, but also the hash of that data. Thus it allows users to verify that the previous block has not been changed. One application of a blockchain is a tamper-evident log. This type of log can only be appended to, since altering earlier changes to logs will be detected.

Now back to the land registry example. This implementation has a fundamental security flaw. For example, Alice has transferred her token to Bob by sending a signed transfer to Bob, but she did not tell anyone else about this transfer. Alice can then create another signed transfer which transfers the same token to Charlie. To Charlie this transfer seems valid, and he is now the owner of the token. Both Bob and Charlie can lay claim to the same land title. This is called a *double-spending attack*, since the same token is being used twice.

To solve this double-spending issue some changes will have to be made to the land registration system.

The first difference is that now the central authority, the Registry, openly publishes an *append-only ledger* with a history of all transfers. Transfers are datastructures with an ID, property ID, buy and seller IDs, and the signatures of all parties. It is also important that a transfer references the previous transfer, as to create a chain of transfers back to the creation of the token. This creates an easy to read chain of transfers and is also easier to validate. The append-only property ensures that anything written to the ledger will remain there forever. This can be used to prevent double-spending by having every transfer written in the ledger before being accepted. This means that it is public knowledge if tokens were sent to a different owner before being sent to an unsuspecting receiver.

The Registry can implement this functionality by using the previously mentioned blockchain datastructure. The chain consists of a series of block with each containing

transfers. Thus, each block will contain the transfer ID, the contents of the transfer, and a hash pointer to the previous block. The registry signs the hash pointer of the head of the chain and publishes its signature along with the blockchain, which finalizes the data in the blockchain.

In this system a transfer is only valid if it is contained in the blockchain published and signed by the Registry. Any user can verify that a transfer is endorsed by the Registry by simply checking the signature of a blockchain containing the transfer published by the Registry. The Registry is careful not to include transfers which double-spend a token.

The blockchain with hash pointers is required to check two users have the history of transactions signed by the Registry. This stems from the append-only property of a blockchain. If some transfers were changed, removed, or added, it will affect all the following blocks due to the hash pointers. Such a change is obvious and easy to catch.

Creating tokens works in the same way, but transferring tokens works differently. Multiple tokens can be exchanged in a single transfer, this requires multiple users to sign the transfer. Thus, the rules for transferring tokens are the following:

- The transferred tokens are valid, which means that they were created in a previous block.
- The transferred tokens are not double-spent, which means they have not already been transferred.
- No new tokens are created, only the Registry can create new tokens.
- The transfer is signed by all the owners of the tokens being transferred.

A transfer is valid if these rules are followed, then the Registry will accept it and add it to the blockchain. Only after the transfer is added to the blockchain and it is published, users can be sure that a transfer has occurred.

This system still has a fundamental problem, however. The problem is the Registry itself, this entity has too much power. It cannot create fake transfers or forge signatures, But it can stop accepting transfers from certain users, thus denying service and making their tokens useless. The Registry can also create an arbitrary number number of tokens if it wants to. Or the Registry can also just give up on the system and publishing new blockchain updates.

The conclusion is that the problem is centralization and the central question is now how to de-Registry-ify the system? Is it possible to get a well-functioning system with this central authority figure?

This is possible if all users can agree on a single public blockchain as the authoritative history of all transfers. The users have to agree on which transfers are valid, which transfers have already occurred to prevent double-spending, and assign IDs in a decentralized way. Creating new tokens also needs to happen in a decentralized way.

Decentralized Systems

This section will discuss on how to move from a centralized system to a decentralized system. Shifting from a centralized blockchain land registration system to a decentralized one involves fundamental changes in how the system is governed and operated. Instead of having a central authority managing, verifying, and recording all transactions, these responsibilities are distributed among multiple participants in the network.

Looking back on the system built in the previous section, there are questions which need to be answered.

- Who maintains and publishes the blockchain?
- Who decides which transfers are valid?
- Who creates more tokens?

Before answering these questions, it is important to discuss what kind of network the system will use. Centralized, decentralized, and distributed networks are different ways in which nodes are connected within a system or network. Each type of network structure has its unique attributes. A visual representation of each structure can be seen in figure 2.1

In a centralized network, all nodes connect to and communicate with a single central node or server. The central node stores data and manages resources, and it is responsible for processing requests and controlling functions within the network.

In a distributed network, all processing power, storage, and functions are spread across all nodes in the network. Every node is equal and capable of performing the same tasks. This is an ideal state where the system is both decentralized (no central point of control) and distributed.

Decentralized networks eliminate the central node and instead distribute control across multiple nodes. Each node operates independently and communicates directly with other nodes.

Since the system requires each node to run independently, the most suitable network structure would be a decentralized network. One such network architecture is a *peer-to-peer*, p2p, network since it is close a purely decentralized network which is completely public.

In this network every node has a copy of the blockchain, each containing a list of blocks which in turn contain a list of transfers. When a Alice wants to transfer a token from to Bob for example, she sends this transfer to all peer-to-peer nodes. When a node receives this transfers, it validates it according to the rules and, if valid, adds it to a set of candidates to add to the blockchain. When there are enough transfers or if a certain time has passed the node attempts to add the transfers to a block and append it to the blockchain.

The problem in this scenario is that every node has its own version of the truth. Multiple users are broadcasting transfers over the network and nodes must agree on the order of the transfers and which ones are already added to the blockchain, thus which transfers are valid. The result is a global blockchain for the entire system. This requires a consensus mechanism.

This results in a system where all nodes in the peer-to-peer network posses a ledger of a sequence of blocks, a blockchain, with each block containing a list of transfers which all nodes have reached consensus on. Each node also has a set of transfers which it has received, but have not been included in the blockchain. This means that consensus has not been reached for these transfers yet, thus this set might be different for each node.

Since peer-to-peer networks are imperfect, nodes might join or leave at any time and nodes might have connectivity issues, and there might be malicious actor, a consensus mechanism in which all nodes must participate is not desirable or even possible. Because the system is also geographically distributed, there might also be a lot of latency which might impair such a consensus mechanism. Remember that public keys are used to identify users and nodes. These can be changed at any time since there is no central authority to assign identities, which makes user identities are non-persistent.

This also makes the design for a consensus mechanism harder since a "one-vote-per-node" is impossible to implement. Malicious actors would simply create a lot of nodes with different identities to acquire an enormous amount of votes, this is called a *Sybil attack*.

Instead the consensus mechanism that the system will use relies on randomness. In essence, the consensus mechanism would run a "lottery" which every node can enter by "buying" tickets. Then a single winner is picked and this winner can propose a new block to be added to the blockchain. This prevents a Sybil attack, because no matter how many identities an actor makes, the actor has to "buy" a ticket. Of course, nodes do not really buy a ticket, but the random selection is approximated by selecting a node in proportion to a scarce resource nobody can monopolize.

This lottery system can be used with different kinds of resources. Below are some of the more popular kinds:

- Computational Power: This is the resource used in *Proof of Work*, PoW, consensus mechanisms. Nodes must use computational power to solve complex mathematical problems, and the first one to find a solution gets to add the next block to the chain. This requires substantial electricity and expensive hardware, making it a significant investment.
- Stake or financial resources: *Proof of Stake* (PoS) and its variations use this type of resource. In these systems, nodes must prove ownership of a certain amount of the network's token. The more a node has at stake, the more likely it is to be chosen to validate the next block.
- Time: Time-locked transactions, used in some Proof of Stake mechanisms, require nodes to lock up their tokens for a certain period, creating an opportunity cost.
- Storage Space: In *Proof of Space* or *Proof of Capacity* consensus mechanisms nodes allocate a certain amount of storage space to the network to participate in the consensus process. This can be a costly resource, as it involves purchasing and maintaining storage hardware.
- Bandwidth: Some consensus mechanisms, like *Proof of Bandwidth*, require nodes to commit a certain amount of network bandwidth to participate in the consensus process.

In the land registry system *Proof of Work* will be used for the sake of simplicity. The idea behind PoW is that the random selection of a node is approximated by basing node selection off of computing power. In essence nodes compete with eachother by using their computing power, which results in nodes being selected in proportion to their capacity.

The system will use hash puzzles to test each node's computing power. When a node wishes to propose a block, it needs to find a number such that when the number is concatenated with the hash pointer and all transactions, then the hash of the total has to be below a certain target. This number is small in comparison to the total output space of the hash function. When the node does propose the block, it needs to add the number as proof of the computation. Because of the avalanche effect, finding this number is difficult. The only way to find the number is brute-forcing, meaning try number one-by-one until the node gets lucky and the hash satisfies the condition.

The difficulty is proportional to the target space size compared to the total output space size. If the target space is 1% of the total output space, then a node has to try 100 numbers to get a number which satisfies the target.

All other nodes can verify this computation easily by simply hashing the number, hash

pointer, and transaction and then comparing it to the proof. If they are equal, the block is valid.

Using this method, there is no "magic" pick a random node function. Instead nodes compete independently until they find a solution and propose a block. Thus the system is completely decentralized.

Below is a step-by-step walk through the system and how it work:

1. Gathering Transactions: as users interact with the system, by registering new properties or transferring ownership of existing ones, their transfers are broadcast to all nodes in the network. Each node collects these transfers into their transfer set, often called a *mempool*. From this set a candidate block of transfers is created.

2. The Lottery: the next step is where the "lottery" comes into play. Nodes "buy" tickets with a scarce resource: computational power. Each node uses its computational power to solve a complex mathematical problem. This problem is designed such that it cannot be solved by simply applying a formula. Instead, the only feasible way to solve it is by making numerous guesses until a solution is found. The node which finds a solution first is the winner of the lottery and can add its candidate block.

3. Adding the Block to the Chain: the winning node broadcasts its block together with the solution. The other nodes can easily verify this solution. Upon verifying that the solution is correct, and the proposed block does not violate any of the network's rules, like double-spending a token, the nodes add the proposed block to their copies of the blockchain. The winner of the lottery is often rewarded with some form of compensation, such as cryptocurrency in a blockchain network or, in the case of our land registry, perhaps a reduction in future transfer costs. This serves as an incentive to take part in the consensus of the system.

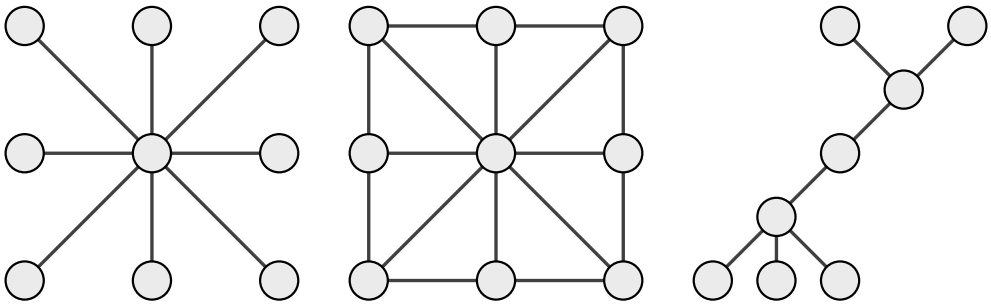
4. Repeat the Process: the process then starts over, with nodes gathering new transactions and participating in a new lottery for the chance to add the next block to the chain. Nodes express their acceptance of the blocks by including its hash in the next block.

   When a node receives two different chains, it simply chooses the longest chain. Two concurrent chains of the same length will eventually resolve itself since nodes are picked randomly and one chain will win out in the end.

This random lottery consensus mechanism makes the network highly secure. As long as no single node controls a majority of the computational power in the network, it is practically impossible for any node or group of nodes to dictate the state of the blockchain.

But what happens if a malicious node is picked for adding a block to the blockchain? Other nodes will continue the chain without the malicious block present, this is an implicit rejection. This makes consensus take longer however, since users might have to wait several rounds before being sure his block is in the "correct" branch of the blockchain.

Consider a denial-of-service attack. Alice decides she does not want to include any transfers coming from Bob's address in any block she proposes. Even though it is a valid attack, the implications are not serious since the proposing node is picked randomly. If Alice gets picked in a round, Bob can wait until the next round to get his transfers into a block.



(a) Centralized network **(b)** Distributed network **(c)** Decentralized network

Figure 2.1: Network architectures.

2.3 Implementing a Blockchain

With an understanding of the theoretical components and operation of a blockchain-based system, the discussion now shifts to the practical aspect of its implementation. This section focuses on the design and implementation of the system using Rust. The choice of Rust for this project is due to its remarkable memory safety guarantees, high performance, and rich ecosystem of libraries.

These sections explores key steps in creating this system, from the basic blockchain data structure to the network side. The aim is to illustrate how theoretical concepts, such as decentralized networks and consensus mechanisms, can be translated into tangible software constructs.

It should be noted that building such a system from scratch involves deep understanding of both the blockchain and Rust programming. Therefore, the following sections will provide a high-level overview, focusing on crucial elements and logic, rather than diving into minute coding details.

The first section examines the foundational component of any blockchain system: the blockchain structure itself.

Basic Blockchain Structure

First fundamental datastructures have to be defined in order to construct a blockchain system,. These datastructures are: the blocks, blockchain, and transfers, here called transactions. In Rust, structs are used to define these data structures. Below are the individual datastructures.

Transaction

A transaction in this context represents a change of ownership of a token representing a land title. It includes details such as the IDs for the current and referenced transaction, property ID, buyer ID, seller ID, and the digital signatures of all involved parties. This can also be represented as a struct, as seen in listing 2.1.

```rust
pub struct Transaction {
    transaction_id: String,
    input_transaction_id: String,
    property_id: String,
    buyer_id: String,
    seller_id: String,
    signatures: HashMap<String, String>, // A mapping of party IDs to their digital
        ↪ signatures
}
```

Listing 2.1: The Transaction struct.

Block

Each block in the blockchain contains its ID, its own hash, a reference to the previous block through its hash, a timestamp indicating when the block was created, a nonce for the PoW consensus mechanism, and finally the list of transactions in the block. Listing 2.2 show this struct in Rust.

```rust
pub struct Block {
    pub hash: String,
    pub prev_hash: String,
    pub timestamp: i64,
    pub nonce: u64,
    pub transactions: Vec<Transaction>
}
```

Listing 2.2: The Block struct.

Blockchain

The blockchain itself is essentially a linked list of blocks. It contains a list of blocks, the difficulty for the consensus mechanism, and the mempool of transactions not yet included in a block. This can be achieved with the struct in listing 2.3:

```rust
pub struct Blockchain {
    pub blocks: Vec<Block>,
    pub difficulty: u32,
    pub mempool: Vec<Transaction>
}
```

Listing 2.3: The Blockchain struct.

The creation of these data structures forms the foundational skeleton of the blockchain system. As the section progresses, these structures will be fleshed out with additional functionality and implement the core features of the system, such as the addition of new blocks and transactions, validation of the blockchain, etc.

Blockchain Operations

With the basic data structures in place, it is possible to proceed to define the operations which interact with the blockchain. These operations include creating new blocks and transactions, adding them to the blockchain, and validating the blockchain.

Creating Blocks

A new block can be created by instantiating the Block struct. The nonce is set to zero, because proof-of-work will be performed on the block later.

```rust
pub fn new(id: u64, prev_hash: String, transactions: Vec<Transaction>) -> Self {
    let timestamp = Utc::now().timestamp();
    let mut nonce = 0;
    let mut hash = "".to_string();

```

```
    Self{id,
         hash,
         prev_hash,
         timestamp,
         nonce,
         transactions,
    }
}
```

Listing 2.4: Function to create a Block.

Hashing Blocks

These functions allow the system to find hashes for blocks below the target value, thus accessing the PoW consensus mechanism The difficulty value tells the functions how many leading zeroes the hash should have.

The `leading_zeros` function counts the leading zeroes of a hash.

The `encode_hex` function converts a vector of bytes to a hexadecimal string for usage in the datastructures.

The `mine_block` function tries to find a hash for a block with a certain amount of leading zeroes. It simply starts from a nonce with value zero and keeps adding one to the nonce until it finds a hash which satisfies the target value.

Finally, the `hash_block` function hashes a block by converting the block datastructure into a vector of bytes and then hashing the result.

```
pub fn leading_zeros(hash: Vec<u8>) -> u32 {
    let mut zeros: u32 = 0;

    for c in hash {
        match c.leading_zeros() {
            0 => break,
            8 => zeros += 8,
            z => { zeros += z;
                   break; }
        };
    }
    zeros
}

pub fn encode_hex(bytes: Vec<u8>) -> String {
    let mut s: String = String::with_capacity(bytes.len() * 2);
    for b in bytes {
        write!(&mut s, "{:02x}", b).unwrap();
    }
    s
}

pub fn mine_block(block: Block, difficulty: u32) -> Block {
    let mut hash: String = encode_hex(block.hash_block());
    let mut tmp_block: Block = Block {
        hash: String::from("0"),
        prev_hash: block.prev_hash,
        timestamp: block.timestamp,
        nonce: block.nonce,
        transactions: block.transactions
    };

    while leading_zeros(hex::decode(&hash).unwrap()) < difficulty {
        tmp_block.nonce += 1;
        hash = encode_hex(tmp_block.hash_block());
    }
```

```
37      info!("Nonce: {}", tmp_block.nonce);
38      info!("{}", hash);
39      tmp_block.hash = hash;
40      tmp_block
41  }
42
43  impl Block {
44      pub fn hash_block(&self) -> Vec<u8> {
45          let mut block_bytes: Vec<u8> = vec![];
46
47          block_bytes.append(&mut bincode::serialize(&self.prev_hash).unwrap());
48          block_bytes.append(&mut bincode::serialize(&self.timestamp).unwrap());
49          block_bytes.append(&mut bincode::serialize(&self.nonce).unwrap());
50          block_bytes.append(&mut bincode::serialize(&self.transactions).unwrap());
51
52          let mut hasher = Sha256::new();
53          hasher.update(block_bytes);
54          hasher.finalize().as_slice().to_owned()
55      }
56  }
```

Listing 2.5: Functions to perform PoW.

Creating Transactions

A new transaction can be created by instantiating the Transaction struct. This is relatively straightforward, but don't forget that the signatures field should be correctly populated with valid signatures from the parties involved.

```
1   pub fn new(input_transaction_id: String, property_id: String, buyer_id: String,
        ↪ seller_id: String, signatures: HashMap<String, String>) -> Self {
2       let id_str = &format!("{}{}{}{}{}", input_transaction_id,
3                                           property_id,
4                                           buyer_id,
5                                           seller_id,
6                                           serde_json::to_string(&signatures).unwrap());
7       let mut hasher = Sha256::new();
8       hasher.update(id_str);
9       let transaction_id =
            ↪ String::from_utf8_lossy(hasher.finalize().as_slice()).to_string();
10      Self {transaction_id,
11            input_transaction_id,
12            property_id,
13            buyer_id,
14            seller_id,
15            signatures,
16      }
17  }
```

Listing 2.6: Function to create a Transaction.

Adding Blocks and Transactions

Blocks can be added to the blockchain by pushing them onto the blocks vector. Transactions that are not yet included in a block can be added to the mempool

```
1   impl Blockchain {
2       pub fn add_block(&mut self, block: Block) -> Result<(), BlockAddError> {
3           let previous_block: &Block = self.blocks.last().expect("There is a least one
                ↪ block");
4           match self.validate_block(&block, previous_block) {
5               Result::Ok(_) =>{
```

```
6                 self.blocks.push(block);
7                 Result::Ok(())
8             },
9             Result::Err(_) => {
10                error!("Couldn't validate block!");
11                Result::Err(BlockAddError)
12            }
13        }
14    }
15
16    pub fn add_transaction(&mut self, transaction: Transaction) {
17        self.mempool.push(transaction);
18    }
19 }
```

Listing 2.7: Function to add Blocks and Transaction to a Blockchain.

Validating the Blockchain

Validating the blockchain involves checking that each block's hash meets the PoW condition and that the previous hash matches the hash of the previous block. Validating transactions has not been implemented yet.

```
1  impl Blockchain {
2      fn validate_block(&self, block: &Block, previous_block: &Block) -> Result<(),
           ↪ BlockValidationError> {
3          if block.prev_hash != previous_block.hash {
4              warn!("block {} has wrong previous hash", block.hash);
5              return Result::Err(BlockValidationError);
6          } else if leading_zeros(hex::decode(&block.hash).expect("Can't decode hex
               ↪ string!")) < self.difficulty {
7              warn!("block {} has invalid difficulty", block.hash);
8              return Result::Err(BlockValidationError);
9          } else if hex::encode(block.hash_block()) != block.hash {
10             warn!("block {} has invalid hash", block.hash);
11             println!("Got {} and expected {}", block.hash,
                   ↪ hex::encode(block.hash_block()));
12             return Result::Err(BlockValidationError);
13         }
14         Result::Ok(())
15     }
16
17     pub fn validate_chain(&self) -> Result<u32, ChainValidationError> {
18         for (block1, block2) in self.blocks.iter().tuple_windows() {
19             match self.validate_block(block2, block1) {
20                 Result::Ok(_) => continue,
21                 Result::Err(_) => {
22                     error!("Couldn't validate block {}!", block2.id);
23                     return Result::Err(ChainValidationError);
24                 }
25             }
26         }
27         Result::Ok(u32::try_from(self.blocks.len()).expect("Can't cast usize to u32"))
28     }
29 }
```

Listing 2.8: Functions to validate Blocks and Blockchains.

Updating the Blockchain

This function updates a blockchain given a second blockchain. Its purpose is to choose between two blockchains when the node receives another blockchain from the network. It always pick the longest valid chain.

```rust
impl Blockchain {
    pub fn update_chain(self, remote: Blockchain) -> Blockchain {
      match (self.validate_chain(), remote.validate_chain()) {
            (Result::Ok(l1), Result::Ok(l2)) => {
                if l1 >= l2 {
                    return Blockchain {blocks: self.blocks, difficulty: self.difficulty,
                        ↪ mempool: self.mempool}
                } else {
                    return Blockchain {blocks: remote.blocks, difficulty:
                        ↪ remote.difficulty, mempool: remote.mempool}
                }
            },
            (Result::Err(_), Result::Err(_)) => panic!("Both chains are invalid!"),
            (Result::Err(_), _) => {
                return Blockchain {blocks: remote.blocks, difficulty: remote.difficulty,
                    ↪ mempool: remote.mempool}
            },
            (_, Result::Err(_)) => {
                return Blockchain {blocks: self.blocks, difficulty: self.difficulty,
                    ↪ mempool: self.mempool}
            }
        };
    }
}
```

Listing 2.9: Function to choose between two blocks.

Network

Once the fundamentals of the blockchain structure and operations are set up, the next step is implementing the network that will support the decentralized system. In a blockchain-based system, the network plays a crucial role in facilitating the decentralization and peer-to-peer interaction.

The blockchain-based land registry operates on a peer-to-peer network, which means each node in the network acts as both a client and a server. Each node in the network holds a copy of the entire blockchain and independently verifies the transactions.

Each node in our decentralized system represents a participant in the land registry system. These could include property owners, brokers, or other relevant authorities. A node should be able to:

- Store the entire blockchain.
- Broadcast valid transactions to other nodes.
- Broadcast new blocks when they are mined.
- Validate and relay transactions and blocks from other nodes.

A logical library choice for this system is `libp2p`. Libp2p is networking framework that enables the development of P2P applications.

Below is the code to setup the node. It uses a command line interface for each node to add blocks, list peers, and interact with the blockchain. As is also visible in listing 2.10, the network uses the gossip protocol instead of flooding.

The network uses three topics: one for blockchain-related message, one for block-related messages, and one for transaction-related messages.

Three command line commands are possible:

- `ls p`: this command show a list of peers.

- `ls c`: this command shows the entire blockchain on the node.
- `create b`: this command adds a block to the blockchain.

In the event loop the node reacts to command line inputs and messages from peers. This code worked fine, but was hard to extend. It was later scrapped, but due to time-constraints the network part of the code was not finished.

```rust
#[async_std::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    let mut new_blockchain: Blockchain = new_blockchain();
    // Create a random PeerId
    let id_keys = identity::Keypair::generate_ed25519();
    let local_peer_id = PeerId::from(id_keys.public());
    info!("Local peer id: {local_peer_id}");

    // Set up an encrypted DNS-enabled TCP Transport over the Mplex protocol.
    let transport = tcp::async_io::Transport::new(tcp::Config::default().nodelay(true))
        .upgrade(upgrade::Version::V1)
        .authenticate(
            noise::NoiseAuthenticated::xx(&id_keys).expect("signing libp2p-noise static
                ↪ keypair"),
        )
        .multiplex(yamux::YamuxConfig::default())
        .timeout(std::time::Duration::from_secs(20))
        .boxed();

    // To content-address message, we can take the hash of message and use it as an ID.
    let message_id_fn = |message: &gossipsub::Message| {
        let mut s = DefaultHasher::new();
        message.data.hash(&mut s);
        gossipsub::MessageId::from(s.finish().to_string())
    };

    // Set a custom gossipsub configuration
    let gossipsub_config = gossipsub::ConfigBuilder::default()
        .heartbeat_interval(Duration::from_secs(10)) // This is set to aid debugging by
            ↪ not cluttering the log space
        .validation_mode(gossipsub::ValidationMode::Strict) // This sets the kind of
            ↪ message validation. The default is Strict (enforce message signing)
        .message_id_fn(message_id_fn) // content-address messages. No two messages of the
            ↪ same content will be propagated.
        .build()
        .expect("Valid config");

    // build a gossipsub network behaviour
    let mut gossipsub = gossipsub::Behaviour::new(
        gossipsub::MessageAuthenticity::Signed(id_keys),
        gossipsub_config,
    )
    .expect("Correct configuration");
    // Create a Gossipsub topic
    let chain_topic = gossipsub::IdentTopic::new("chain");
    let block_topic = gossipsub::IdentTopic::new("block");
    let transaction_topic = gossipsub::IdentTopic::new("transaction");
    // subscribes to our topic
    gossipsub.subscribe(&chain_topic)?;
    gossipsub.subscribe(&block_topic)?;
    gossipsub.subscribe(&transaction_topic)?;

    // Create a Swarm to manage peers and events
    let mut swarm = {
        let mdns = mdns::async_io::Behaviour::new(mdns::Config::default(),
            ↪ local_peer_id)?;
        let behaviour = MyBehaviour { gossipsub, mdns };
        SwarmBuilder::with_async_std_executor(transport, behaviour, local_peer_id).build()
    };

```

```
    // Read full lines from stdin
    let mut stdin = io::BufReader::new(io::stdin()).lines().fuse();

    // Listen on all interfaces and whatever port the OS assigns
    swarm.listen_on("/ip4/0.0.0.0/tcp/0".parse()?)?;

    // println!("Enter messages via STDIN and they will be sent to connected peers using
        ↪ Gossipsub");

    // Kick it off
    loop {
        select! {
            line = stdin.select_next_some() => {
                match line.expect("stdin not to close").as_str() {
                    "ls p" => handle_print_peers(&swarm),
                    cmd if cmd.starts_with("ls c") => handle_print_chain(&new_blockchain),
                    cmd if cmd.starts_with("create b") => handle_create_block(&mut
                        ↪ new_blockchain),
                    _ => error!("unknown command"),
                }
            },
            event = swarm.select_next_some() => match event {
                SwarmEvent::Behaviour(MyBehaviourEvent::Mdns(
                        mdns::Event::Discovered(list))) => {
                    for (peer_id, _multiaddr) in list {
                        info!("mDNS discovered a new peer: {peer_id}");
                        swarm.behaviour_mut().gossipsub.
                            add_explicit_peer(&peer_id);
                    }
                },
                SwarmEvent::Behaviour(MyBehaviourEvent::Mdns(
                        mdns::Event::Expired(list))) => {
                    for (peer_id, _multiaddr) in list {
                        warn!("mDNS discover peer has expired: {peer_id}");
                        swarm.behaviour_mut().gossipsub.
                            remove_explicit_peer(&peer_id);
                    }
                },
                SwarmEvent::Behaviour(MyBehaviourEvent::Gossipsub(
                       gossipsub::Event::Message {
                    propagation_source: peer_id,
                    message_id: id,
                    message,
                })) => info!(
                        "Got message: '{}' with id: {id} from peer: {peer_id}",
                        String::from_utf8_lossy(&message.data),
                    ),
                SwarmEvent::NewListenAddr { address, .. } => {
                    info!("Local node is listening on {address}");
                }
                _ => {}
            }
        }
    }
}
```

Listing 2.10: Code to setup the node.

Below are functions to handle specific events on the node. These react to the command line inputs of the user.

```
pub fn get_list_peers(swarm: &Swarm<MyBehaviour>) -> Vec<String> {
    info!("Discovered Peers:");
    let nodes = swarm.behaviour().mdns.discovered_nodes();
    let mut unique_peers = HashSet::new();
    for peer in nodes {
```

```
6           unique_peers.insert(peer);
7       }
8       unique_peers.iter().map(|p| p.to_string()).collect()
9   }
10
11  pub fn handle_print_peers(swarm: &Swarm<MyBehaviour>) {
12      let peers = get_list_peers(swarm);
13      peers.iter().for_each(|p| info!("{}", p));
14  }
15
16  pub fn handle_print_chain(chain: &Blockchain) {
17      info!("Local Blockchain:");
18      let pretty_json =
19          serde_json::to_string_pretty(&chain.blocks).expect("can jsonify blocks");
20      info!("{}", pretty_json);
21  }
22
23  pub fn handle_create_block(chain: &mut Blockchain) {
24      let new_block = Block {
25              id: u64::try_from(chain.blocks.len()).expect("usize not bigger than u64"),
26              timestamp: Utc::now().timestamp(),
27              prev_hash: chain.blocks.last().expect("at least one previous
                    ↪ block").hash.clone(),
28              nonce: 0,
29              hash: String::from("0"),
30              transactions: vec![]
31      };
32      let mined_block: Block = mine_block(new_block, chain.difficulty);
33      match chain.add_block(mined_block) {
34          Result::Ok(_) => info!("Successfully added block!"),
35          Result::Err(_) => panic!("Couldn't add block!")
36      };
37  }
```

Listing 2.11: Functions to handle events in the node.

Now that the basics of blockchain have been explained, the reader is ready to dive more in-depth into modern blockchain technologies. The following chapters discuss Bitcoin, Ethereum, and finally Cardano.

Chapter 3

Bitcoin

Bitcoin is a collection of technologies similar to the prerequisites discussed in chapter 2. Instead of recording land transactions and registrations as shown in chapter 2, it stores and transmits a different value. This value is a currency also called Bitcoin on the Bitcoin network. The network runs primarily on the Internet but can also use other transport networks.

Participants can use Bitcoin on the network to do almost anything that can be done with conventional currencies. The activities include buying and selling goods, sending money to people and organizations, and extending loans.

The difference is that, unlike conventional currencies, Bitcoin is entirely virtual. There are no physical or digital coins. The "coins" are implied in transactions that transfer data from address to address. One address is the sender, and the other address is the recipient. The addresses are the keys that allow participants to prove ownership of Bitcoin on the network. Participants use these keys to sign transactions, unlocking the value and allowing them to spend it by transferring them to new addresses and, thus, a new owner. Owning a valid key to sign a transaction is the only requirement for spending Bitcoin. The participant has complete control of his Bitcoin.

Bitcoin is a distributed, peer-to-peer system. There is no central authority or point of control, like a bank, which controls transactions or mints new currencies. New Bitcoin is created by "mining", which will be discussed later in this chapter. This mining process is also its consensus algorithm. Any participant who runs a full node called a protocol stack can mine Bitcoin. In principle, Bitcoin mining decentralizes currency issuance, thus taking over the responsibilities of a traditional central bank.

The protocol includes algorithms that regulate the mining functions of the network. These algorithms ensure that the time to mint new currency stays constant. It will always take around ten minutes to mint new Bitcoins. The protocol also halves the amount of Bitcoin mined every four years. The protocol also limits the total supply of Bitcoin to a fixed 21 million coins. The Bitcoin in circulation follows an elliptic curve that approaches 21 million by 2140. This issuance model means the currency is deflationary, and it is impossible to inflate Bitcoin by printing more currency at will.

Bitcoin is also the name of the protocol. The Bitcoin currency is merely an application of this protocol. It represents the culmination of four key innovations. These were already introduced in chapter 2 but will be nuanced and explained in the context of Bitcoin in this chapter. These four innovations are:

- The Bitcoin protocol: a decentralized peer-to-peer network.

- The blockchain: a public and decentralized ledger.
- Consensus rules: rules for independent transaction validation.
- The consensus algorithm: a mechanism for reaching globally decentralized consensus.

3.1 Overview

This section gives a high-level overview of the Bitcoin blockchain before jumping into more technical details. This section introduces the general concepts, jargon, and purpose of Bitcoin and its mechanisms.

What purpose does Bitcoin even serve? Before Bitcoin, it was possible to relay transactions using networks. The problem, however, is that malicious actors can insert conflicting transactions in the network. For example, the actor can create two transactions spending the same value simultaneously. This action is called a double spend.

Nodes

The Bitcoin network consists of every person running the Bitcoin software, also known as clients or nodes. These clients communicate with each other about the state of the network. This communication is necessary to keep every node up-to-date.

A node has three jobs in the network: follow the protocol, share information, and keep a copy of confirmed transactions.

Each node has been programmed to follow the rules constituting the Bitcoin protocol. A node uses these rules to check the transactions it receives, and the node can only relay transactions if they comply. If there are any violations, they will not be passed on to the rest of the network. For example, one rule is that a transaction cannot spend more in the output than the sum of the inputs. So if the node does receive a transaction where the sum of the outputs is larger than the inputs, the node will not pass on the transaction to other nodes and will blacklist the sender.

The most critical information a node passes on is transactions. There are two main types of transactions in the Bitcoin protocol:

- Fresh transactions which have recently entered the network.
- Confirmed transactions have been confirmed and entered into the blockchain. These are received in blocks rather than individually.

The difference between these two is discussed later in this section when mining and blocks are discussed.

New transactions spread around the network until they are finalized into the blockchain. Each node has a copy of the blockchain for reference and will share it with others if their copies are not current. The process of adding new transactions to the blockchain is called mining.

Each node is an autonomous entity. The network does not tell a node what to do. The node knows what it has to do and makes decisions on its own. This information means that the entire network is decentralized.

It is also important to note that a Bitcoin user does not have to run a node to send or receive Bitcoin. A user only has to get a transaction into the network by sending a message to a node.

Mining

Mining is the process of adding transactions to the blockchain. As mentioned in the previous section, nodes share new transactions over the network. Nodes then store these transactions in memory; this is called the mempool. Each node can try and mine transactions in its mempool into the blockchain. A node has to use a lot of processing power to add transactions to the blockchain.

This processing power is forced onto the node by a challenge. A node has to hash a list of transactions and attempt to get a hash value prefixed by several zeroes. The number of zeroes indicates the difficulty set by the network and the number of miners in the network. The node can hash the transaction with another number to get different results; this number is called a *nonce*.

This sound easy, but computationally this is difficult and random. The only way to get a valid result is by trial and error. This action is what mining is, a lot of hashing and hoping to get lucky.

When a node gets lucky enough, its selected transactions get added to the blockchain, and every other node adds the created block to its blockchain. The node also gets a reward for its effort and any transaction fees.

One question remains: why not just add transactions to the blockchain directly? Mining allows the network to agree on which transactions to add; this is called consensus.

Take the double-spending example from the introduction. When a user creates a Bitcoin transaction, only some nodes receive the transaction instantly. Instead, transactions get gossiped around the Bitcoin network. Creating another transaction that spends the identical Bitcoin and inserting it into the network is possible. Even though the second transaction is created after the first transaction, the Bitcoin network would disagree about which transaction to accept because of how transactions travel across the network.

The network decides which transaction to pick by using mining. When a node completes the challenge, its picked transactions get added to the blockchain. It takes ten minutes for each new block of transactions to be added to the blockchain. This method seems unorthodox and random to reach a consensus, but it does work.

Blockchain

Every node on the Bitcoin network shares a copy of the blockchain. The copy is necessary because it allows nodes to have a complete list of transactions, and the nodes can then work out how many Bitcoin are located at each address. This system is reminiscent of a logbook or ledger.

This data structure is called a blockchain because transactions are not added individually but grouped in blocks. Grouping transactions into blocks makes it easier to propagate changes in the blockchain. Instead of updating the shared data structure several times per second, the nodes group transactions into blocks and update the data structure once every ten minutes. These blocks are linked in a chain so that any change in lower blocks is propagated up to the top. This chain makes tampering with blocks or transactions more difficult without other nodes noticing.

Nodes on the Bitcoin network share the blockchain. This network is a peer-to-peer network.

A user can get a copy of the blockchain by downloading the Bitcoin client. Once installed and executed, the client connects to the network and downloads the blockchain. As of writing, the size of the blockchain is 486.77 GB; thus, it will take some time. The

blockchain is this large because it contains every Bitcoin transaction since the third of January 2009. Downloading the blockchain is a one-off occurrence. Once this is finished, only individual blocks need to be received, which are only 1 MB in size.

An interesting question is what happens when two branches have the same length. This phenomenon is called a chain split. The nodes will be network-partitioned until one chain becomes dominant. Nodes will handle the equal branches on a first-come, first-serve basis. The method means there will be a network partition where some nodes assume chain A is the canonical chain, and others assume chain B is the canonical chain. In such an event, miners race to produce a new block to make their canonical chain the longest chain first and broadcast it to the network to achieve convergence. When the contention is resolved, all nodes will switch to the new longest chain and obsolete the old chain.

Blocks

A block is a set of transactions added to the blockchain. Blocks are created by miners, as mentioned previously. When a user creates a Bitcoin transaction, it is not added immediately. A new transaction is held in the mempool before being processed. Miners gather transactions from the mempool into a candidate block and then try to add the candidate block to the blockchain.

Each block has a block header. This block header consists of metadata about the block:

- Version: Version of the Bitcoin protocol used by the block.
- Last block: The identification number of the previous block.
- Transaction root: The transactions are in a Merkle tree data structure. The root points to the root of the tree.
- Time: Current time.
- Target: Value the miners work with to add blocks to the blockchain. The network sets this value.
- Nonce: Free value that can be adjusted to hit the hash target.

Miners use the header when attempting to add the block to the blockchain.

To add a block to the blockchain, a miner hashes the block header and hopes the resulting value is below a specific value. The target value is dictated by the difficulty, which is set by the Bitcoin network to regulate of complex it is to add a block of transactions to the blockchain. The difficulty is designed to regulate how quickly blocks are solved to set the time between blocks added to the blockchain to approximately ten minutes. The delay is essential to add blocks regularly, even as more miners join the network. If the difficulty remains the same, the time to add blocks will reduce over time. The difficulty adjusts every 2016 block, which is roughly two weeks. When this interval is reached, every node divides the expected mining time for the 2016 blocks by the actual time. If this number is more significant than one, the difficulty increases. If it is less than one, the difficulty decreases.

The nonce is a free field that miners can use to try and get the hash value below the target value. Said in different way, the hash value has to start with a certain number of zeroes. Nonce means “number used only once” and can be any arbitrary number. If the first nonce does not work, a miner keeps incrementing the nonce until a good hash is hit.

Once a nonce is found, the miner can add the block to the blockchain and propagate the new blockchain through the network. If the new blockchain is well formed and suffices,

other nodes will accept the updated blockchain as the new blockchain. All miners will now look back at their mempool, pick new transactions to put into the candidate block and start working on the new block. The miners will use the successfully added block as the new block hash in the header and repeat the race.

Transactions

A transaction contains information about a value being sent to one address and an amount spent from this address to another. A Bitcoin address keeps track of each transaction it has received. Different from traditional account-based models, like in banking, users can only take an approximate amount of Bitcoin to spend somewhere. They have to spend Bitcoins in batches.

The data contained in a transaction is easily represented in a single line of data. A user sends this data line into the Bitcoin network when he creates a transaction. Eventually, one of the nodes in the network will mine this transaction into a block; this block will be added to the blockchain and marked as confirmed. That is all there is to a Bitcoin transaction: feeding a simple line of data into the Bitcoin network and waiting for it to be mined into the blockchain.

When a user wants to send Bitcoin to someone else, he has to use the whole amount he has received and send the Bitcoin to a new address. In essence, users receive batches, called outputs, of Bitcoin and must spend whole outputs simultaneously. What if these outputs add up to more than a user wants to spend? In this case, the user can add another output to a transaction where he sends the difference to himself. The model is called the UTXO model, which stands for "Unspent Transaction Output".

Consider the example illustrated in figure 3.1: a user wants to buy an item costing 11.2 Bitcoin. He does not have a single output to his address which covers the cost of 11.2 Bitcoin. Instead, he gathers a collection of outputs with a total greater than 11.2 Bitcoin, in this case, 12 Bitcoin. This collection of outputs is referred to as the inputs of the transaction. Using the total input of 12 Bitcoin, the user creates two new outputs: 11.2 Bitcoin to the item's seller and less than 0.8 back to his address. The difference between the inputs and outputs is called transaction fees. These fees are gathered by miners when they mine a block; this creates an incentive for miners to include the transaction in a block.

The outputs used as inputs for the transaction are considered spent and cannot be used again. The new and unused outputs for the address are unspent and can be used freely, so the model is called "Unspent Transaction Output".

In summary:

- A user has a Bitcoin address. Bitcoin arrives at this address in batches, called outputs.
- A Bitcoin transaction is the process of using these outputs as inputs of another transaction to create new outputs which belong to other users.
- This can be represented as a single line of data when encoded.

Other users cannot spend a user's Bitcoin because each transaction output has an output lock. Nodes will reject the transaction if a user creates a transaction with another user's transaction output and cannot unlock the output. Every user with a Bitcoin address also has a private key, sometimes called a secret key. If a user wants to spend a transaction output, he can use this private key to unlock the output and send it to another user. After the transaction is created and the output into the transaction is unlocked, all nodes will accept the transaction and propagate it throughout the Bitcoin network.

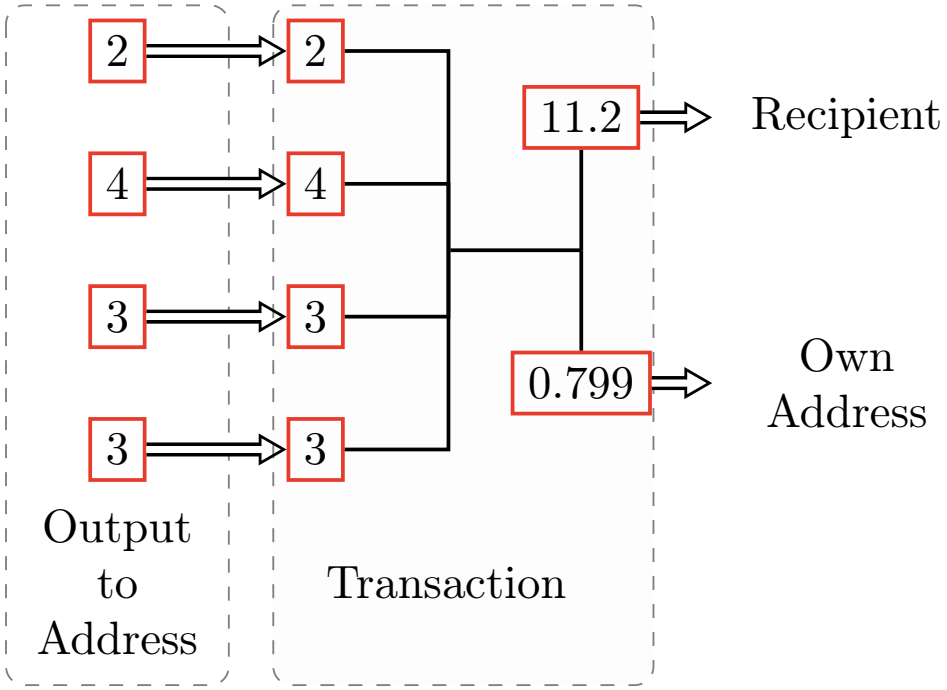


Figure 3.1: A simple UTXO example.

Do note that the unlocking sequence described in the paragraph above is only one way of unlocking a UTXO. Checking the identity is just one instruction in a special purpose programming language described in section 3.4.

An output lock is a set of requirements placed on an output. To spend these outputs, a user must satisfy these requirements. The most common type of lock locks an output to an address and requires a specific private key to unlock it. These locks prevent users from spending each other's outputs. During the creation of a transaction, the creator can put a lock on outputs. When a user wants to send Bitcoin to someone, he can add a lock saying, "Only the owner of key X can use this output". The output will belong to the recipient because they are the sole owner of the private key for this address when he uses the lock.

In essence, Bitcoin is never really sent. Instead, users construct transactions that create outputs and send these transactions to the Bitcoin network to get mined. The blockchain is a data structure of transactions, but in practice, one can think of it as a storage for outputs. When a user wants to send Bitcoin to someone, he can refer to unlockable outputs in the blockchain. When this transaction gets mined, the used outputs become unspendable.

Output locks and unlocks are written in a programming language called *Script*. The locks are discussed in depth in section 3.4.

3.2 Addresses

Ownership of Bitcoin is achieved through keys, addresses, and signatures. Users create the keys. They can be saved in a file, database, or something else. Note that these are entirely separate from the Bitcoin protocol. These keys enable many exciting properties, including proof of ownership, mathematical security, and decentralized trust.

These aspects are essential. Signatures are required to prove that a user can spend his Bitcoin. So only a specific user must have access to his private key as proof of identity.

All transactions require a valid digital signature generated by a private key to be considered for inclusion on the blockchain. This property means that everyone with access to the private key can claim ownership of the Bitcoin included in those transactions. A common term in cryptography is *witness*. The term refers to the signature used to spend funds. In a Bitcoin transaction, the witness data attest to the actual ownership of the funds.

The keys come in pairs: a private key and a public key. One can think of the public key as a public identity and the private key as a secret PIN or signature. The private key provides control over the Bitcoins.

When a transaction is created, the address of the recipient is required. This address is not simply the recipient's public key but is generated from the public key using a one-way function. The 160-bit hash function can be seen in figure 3.2. The public key is analogous to the beneficiary's name. Usually, the Bitcoin address is generated from the public key. However, this is only sometimes the case. Sometimes the Bitcoin address represents a different kind of beneficiary, such as scripts. A script can simply check the ID of the beneficiary, but can also be something much more complex. These scripts abstract the recipient of Bitcoin, making transaction output more flexible.

These addresses also include a checksum to prevent users from accidentally mistyping the address and losing Bitcoin permanently. This checksum is derived by hashing the address using SHA-256 twice and taking the first four bytes of the resulting hash. This checksum is then used to check if an address is valid when a user needs to use it somewhere.

Asymmetric Cryptography

Public key cryptography was invented in the 1970s and is still the mathematical backbone of computer security. Since its invention, multiple suitable functions have been discovered. These functions are practically irreversible. The public key is easy to generate from the private key and verify. However, knowing the public key, it is infeasible to compute the private key. This scheme allows for the creation of digital secrets and signatures based on these properties. Both of these are unforgeable. Bitcoin uses an elliptical curve function as its basis for cryptographic security.

Public key cryptography is used to generate a key pair. The public key is used to receive funds, and the private key is used to sign transactions and spend funds. A mathematical relationship between the two keys allows the private key to generate message signatures. This signature can then be validated with the public key without revealing the private key. If the signature is valid, one can assume the signature originated from the owner of the private key.

When a user wants to spend Bitcoin, he presents his public key and a signature in a transaction to spend that Bitcoin. With this information, everyone in the Bitcoin network can verify that the transaction is validity. The validity confirms that the user who wants to transfer the Bitcoin owns them.

The private key is a number, usually chosen at random. We can use elliptic curve multiplication to generate a public key from this private key. The operation is a one-way cryptographic function. To get a Bitcoin address, a user uses another one-way cryptographic function, a hash function. This section will discuss the private key generation, delve into elliptic curve mathematics to turn it into a public and generate a Bitcoin address from this result. Figure 3.3 shows this relationship.

Private keys

The private key must remain a secret at all times. If a third party gains access to private keys, it is identical to giving them control of the Bitcoin corresponding to that key. Protection from accidental is also necessary. Because once a private key is lost, there is no means of recovery, and funds secured by this key are lost too.

As mentioned, private keys are generated by picking a random number. Thus, the first step is finding a secure and reliable entropy source. The Bitcoin core implementation used the OS's random number generator to produce 256 bits of entropy. The random number can be any number from 1 to $n-1$, where n is a constant defined as the order of the elliptic curve. In this case, slightly less than 2^{256}. A 256-bit number is generated to create this key, then check if it is less than $n-1$. Usually, this is achieved by feeding random bits gathered from the previously mentioned random source into the SHA256 hash algorithm. The SHA256 algorithm always produces a 256-bit number. If the result satisfies the condition, the private key has been generated.

Elliptic Curve Cryptography

Elliptic curve cryptography is based on the discrete logarithm problem. Bitcoin uses a specific elliptic curve and constants defined in the *secp256k1* standard. The *secp256k1* curve is defined by the following function: $y^2 = \left(x^3 + 7\right)$ over $\mathbb{F}_p$ or $y^2 \bmod p = \left(x^3 + 7\right) \bmod p$. The modulo p indicates that this curve is over a finite field of prime order p. The formula can also be written as $\mathbb{F}_p$. Here is $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, a very large prime number. This curve can be seen in figure 3.4.

Note that this is not an actual curve but more like points scattered on a curve because the curve is defined over a finite field. The mathematics stays the same compared to an actual curve.

In elliptic curve mathematics, there is a concept called the point-at-infinity. The concept acts as the identity element in addition. There is also a + operator called addition. Its properties are the same compared to addition in real numbers. Given two points, P_1 and P_2, there exists a third point, $P_3 = P_2 + P_1$, which is also on the elliptic curve.

Geometrically, this third point is calculated by drawing a line between the two other points. This line intersects the curve at exactly one point, $P_3' = (x, y)$. Then mirror over the x-axis to get $P_3 = (x, -y)$.

Some exceptional cases require the point-at-infinity. For example, if P_1 and P_2 have the same x-values, the line between them will be vertical. In this case, the point P_3 is at infinity.

Addition is associative. Now that we have an identity element and addition is defined, we can define multiplication in the standard way. For a point P on the elliptic curve, if k is an integer. Then $k \cdot P = P + P + \cdots + P$ (k times). The integer k is sometimes called the exponent, which might be confusing.

Public keys

To generate public keys from private keys, elliptic curve multiplication is used. The multiplication looks like this: $K = k \cdot G$, where K is the resulting public key, k is the known private key, and G is the generator point, a constant defined by the elliptic curve. The reverse operation finds the discrete logarithm. The operation is a brute-force search. Figure 3.5 show how a public key is calculated.

This multiplication is also known as a trap-door function in cryptography. It is easy to perform in one direction but impossible in reverse.

The generator point is the same for every user. Thus, the private key multiplied by the generator point will always have the same public key. This relationship is fixed.

Addresses

Bitcoin addresses are derived from previously generated public keys. It uses another one-way cryptographic function called a hash function. This function produces a fingerprint from an arbitrary-sized input. The hashing uses two algorithms: SHA256 and RIPEMD160. The SHA256 hash is computed starting from the public key K. The computation results in a 256-bit hash, and then the RIPEMD160 hash is computed, which in turn results in a 160-bit hash: $H = \text{RIPEMD1}(\text{SHA256}(K))$.

The address is not represented in its binary form. It is encoded as a Base58 number. Base58, in essence, is Base64 without the zero, O (capital case o), l (lower case L), I (capital case i), and "+" and "/". The goal of Base58 is to reduce the number of transcription errors.

3.3 Transactions

A transaction is the most essential aspect of the entire Bitcoin system. Everything revolves around ensuring transactions can be created, circulated, validated, and added to the ledger securely. In essence, transactions are data structures that encode value transmission between parties. Each transaction is a public record on the blockchain.

This section examines the contents of transactions, how they are created and verified, and finally, how they become permanent entries.

Transaction Structure

Here the contents and structure of transactions are examined. The contents of a raw transaction are shown in listing 3.1.

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
  {
    "txid":
    "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d8
    1de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
    "3045022100884d142d86652a3f47ba4746ec
    719bbfbd040a570b1deccbb6498c75c4ae24cb02204
    b9f039ff08df09cbe9f6addac960298cad530a863ea
    8f53982c09db8f6e3813[ALL]0484ecc0d46f1918b3
    0928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe4
    23cc5412336376789d172787ec3457eee41c04f4938
    de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
  ],
  "vout": [
  {
    "value": 01500000,
    "scriptPubKey": "OP_DUP OP_HASH160
    ab68025513c3dbd2f7b92a94e0581f5d50f654e7
    OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
    "value": 08450000,
```

```
30      "scriptPubKey": "OP_DUP OP_HASH160
31      7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
32      OP_EQUALVERIFY OP_CHECKSIG",
33    }
34    ]
35  }
```

Listing 3.1: Example of a Bitcoin transaction

Some crucial parts need to be included. There is no explicit address field or input value. All these concepts are constructed on a higher level.

The fundamental building blocks of transactions are transaction outputs. Transaction outputs are chunks of Bitcoin currency, are recorded on the blockchain, and are recognized as valid by the network. Bitcoin nodes track all spendable outputs. Spendable means that an input script can unlock the output of a transaction. These outputs are known as unspent transaction outputs, abbreviated as **UTXO**, and this model is called the UTXO model. The entire collection of all UTXO is called the UTXO set; currently, there are millions. As more UTXOs are created, the set grows and shrinks as they are consumed. These changes can be seen as state transitions.

It might be puzzling that transactions are done in this type of model. Why can users not simply create a transaction which says: "Send x amount from address yy to address zz" and sign it to verify they own the account? Creating such a transaction is undoubtedly possible, and another blockchain, Ethereum, discussed in chapter 4, uses this model. This model is called an account-based model. Both models have their advantages and disadvantages. Bitcoin uses the UTXO model because it is more transparent and auditable. Tracing a value's origin is easy by tracing the transaction chain. It is also deterministic since each transaction has a clear input and output. The account-based model can suffer from overdrawn accounts and chargebacks, Accounts are also a source of global state; the account database has to be updated each time a block is added. UTXO, on the other hand, can be seen as a directed acyclic graph in which each node is a transaction. Another benefit is that UTXO can be processed in parallel, while transactions between accounts must be processed sequentially.

In common parlance, users say a wallet "receives" funds. Receiving funds means the wallet has detected a UTXO that it can spend with its keys. The implication is that a user's "balance" is the sum of all UTXOs that his wallet can spend. The balance can be scattered among an arbitrary number of transactions and blocks. As written in the previous paragraphs, the wallet application constructs the concept of a "balance" at a higher level here. The wallet scans the blockchain and summarizes the UTXO values it can spend.

A fraction of a Bitcoin represents the actual transaction output, a satoshi. A fraction can be an arbitrary integer value. Regular currencies can be divided into two decimal places, called cents, and Bitcoin can be down to eight decimals, called satoshis. Once the output is created, it can no longer be divided. The division is an important characteristic: outputs are discrete and indivisible values denominated by integers, also called satoshis. Output has to be consumed in its entirety by another proceeding transaction.

If a UTXO is larger than desired, it must still be consumed. Now the new transaction has a surplus. For example, take a UTXO with 20 Bitcoin as an output, and a user wants to spend one. The user has to consume the entire UTXO worth 20 Bitcoin and generate two outputs. One pays one Bitcoin to the intended recipient, and the other 19 surpluses back to his address.

As in real life, the user must make choices to purchase. Does he look for exact change by combining several UTXOs, consuming a larger one, and creating a surplus? All

this complexity is handled by wallet software but is necessary when a developer want to construct transaction programmatically from UTXO.

This model creates a chain of transactions in which each consumes previous UTXOs. This way, Bitcoin values move from user to user in this chain. Now the question remains of how new Bitcoin values are created. A unique transaction is called a **coinbase transaction**. A coinbase transaction is the first transaction in each block. The creator of the block can insert it here and creates a new Bitcoin payable to that creator. The transaction does not consume any UTXO but has a unique coinbase input. The coinbase transaction is how the new Bitcoin supply is created during mining.

A transaction output consists of two parts, which can again be seen in listing 3.1:

- An amount of Bitcoin denominated in satoshis and the "`value`" field.
- An output script which determines the conditions for spending the output, as can be seen in the "`scriptPubKey`" field.

This condition is also called the *locking script*, *witness script*, or *scriptPubKey*.

The topic of unlocking and locking scripts and the scripting language is discussed later in section 3.4. More information on the transaction structure is needed before these topics are discussed.

Transaction inputs identify which UTXOs will be consumed by the transaction. They also provide proof of ownership by providing an unlocking script. For each UTXO the transaction points to, the transaction creates an input and has to unlock it with an unlocking script.

Listing 3.2 shows only the transaction input. The field "`txid`" contains the transaction hash of the referenced UTXO, and the "`vout`" field is the output index of the UTXO in that transaction. Together they point to a single UTXO on the blockchain. The second part, which continues the locking script, is called the unlocking script, here the "`scriptSig`" field and simply a constant. The unlocking script has to satisfy the spending requirement of the UTXO referenced by the input.

```
"vin": [
{
  "txid":
  "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d8
  1de548f0a65a8a999f6f18",
  "vout": 0,
  "scriptSig" :
  "3045022100884d142d86652a3f47ba4746ec
  719bbfbd040a570b1deccbb6498c75c4ae24cb02204
  b9f039ff08df09cbe9f6addac960298cad530a863ea
  8f53982c09db8f6e3813[ALL]0484ecc0d46f1918b3
  0928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe4
  23cc5412336376789d172787ec3457eee41c04f4938
  de5cc17b4a10fa336a8d752adf",
  "sequence": 4294967295
}
]
```

Listing 3.2: Example of a transaction input

This specific input only contains a single UTXO because it contains sufficient funds for the payment. Thus, the input points to transaction `"txid": 7957a35f . . . 6f18` and output index `0`, the first UTXO created by the transaction. Moreover, it contains an unlocking script that satisfies the UTXO "`scriptPubKey`". Once this transaction is broadcast to the network, every validating node needs to retrieve the referenced UTXO to validate the transaction.

A validating node is a node that checks if every transaction, both fresh and already confirmed in a block, it receives is valid. Valid means that they are balanced, the input is greater or equal to the output, and everything within the block is correct. This validation allows the network to trust blocks without trusting the miner who created them.

The input values, the missing value, and the address create extra effort when creating programs for handling Bitcoin transactions. For example, creating a simple program that calculates the fees paid requires extra steps. This program calculates the difference between input and output values. However, because of the way inputs are represented, the program needs to fetch output values from the previous UTXO referenced in the inputs. Some extra steps and transactions are needed because of this missing context.

Transaction fees are a way to compensate miners for securing the network. They are also a security measure in and of itself. It makes it infeasible for attacks to flood the network with transactions because the cost would be too high.

A higher fee makes it more likely for a transaction to be included in the next block. The fee is collected by the miner who created the block, which includes the transaction. Fees are based on the transaction size, not necessarily the value within the transaction. Miners prioritize transactions based on many criteria, but transactions with sufficient fees are generally prioritized. Transactions with inadequate or no fees are processed on a best-effort basis and might be delayed.

Over time the fee policy has evolved. At first, they were fixed across the network, but gradually the policy relaxed and has been influenced by market forces, capacity, and volume. Currently, zero and low-fee transactions will not get mined and sometimes not even propagated. Individual nodes set these fee relay policies and can be set to an arbitrary value.

Fees

When transactions are created, the programmer must implement dynamic fee estimations. The program can use a third-party estimator service or implement an algorithm itself. The fee is calculated based on the current network capacity and fees set by competing transactions. The algorithm can be simple, for example, just calculating the median fee, or more complicated by statistical analysis. The goal is to optimize the probability of being included in a block as fast as possible with the minimum fee.

Below is listing 3.3, an example of a third-party service can be seen. The service is a simple API that returns a JSON object with recommended fees in three tiers. These are expressed in satoshis per byte. With `fastestFee` resulting in the fastest transaction confirmation, 0 to 1 blocks, and `hourFee` will confirm the transaction within the hour, or six blocks, with a 90% probability.

```
1 $ curl https://Bitcoinfees.earn.com/api/v1/fees/recommended
2
3 {"fastestFee":102,"halfHourFee":102,"hourFee":88}
```

Listing 3.3: Example of a third-party transaction fee estimation service

As mentioned before, the fee of a transaction is calculated by taking the difference between the sum of the transaction inputs and the sum of the transaction outputs. It is essential to remember this to avoid accidentally including huge fees by under-spending inputs. The creator must always account for all inputs and, if needed, create a surplus into his address.

3.4 Script

The script language used by Bitcoin scripts also called *Script*, is a Forth-like RPN language. It is Forth-like in the sense that it is stack-based. RPN means Reverse Polish Notation, which means operators are postfixed. The notation is the reverse of Lisp-like languages, where the operator is prefixed. For example, $1 + 2$ is written as 12+. Both the locking and unlocking scripts of a UTXO are written in this language.

The script is a deliberately simple language and has a limited scope. Multiple attempts have been made to make it more expressive and extend the scope, but this has always resulted in many problems and exploits. Such extensions have also happened with Ethereum and will be discussed more extensively in chapter 4. Its simplicity is a security feature when used for validating programmable money.

Most transactions are "Payment to `x's` Bitcoin address". This kind of script is called a *Pay-to-Public-Key-Hash* script. However, they are open to more than this scheme and can express several more complex conditions. Before addressing more complex scripts, the basics of Script and transaction scripts will be discussed.

While the language contains many operators, one might notice that it is limited in one crucial way: there are no loops or complex control flow capabilities other than conditional control flow. These limitations make the Script *Turing incomplete*. The limitations put on the language ensure the scripts have predictable execution times, which puts less of a burden on the network. A malicious actor cannot create infinite loops or embed any logic bomb to cause a denial-of-service attack. These limitations ensure not to turn the transaction validation mechanism into a vulnerability.

Another critical aspect of the language is being stateless. Statelessness is a result of the UTXO model. The language only needs the locking and unlocking script of the referenced and current transaction. No previous state is needed, and the result must not be saved after execution. Thus, all information needed to execute a script is contained within that script. Scripts are, therefore, also deterministic. It will consistently execute when run with the same outputs and inputs and have the same result. The same is true when verifying the script; if one node verifies a script, it can be sure that all other nodes will come to the same conclusion. This predictability is an enormous benefit for the Bitcoin network.

As previously mentioned, the validation engine needs two types of scripts: the locking and unlocking script. The locking script places a condition on the output, and the unlocking script solves or satisfies this condition and allows the output to be used. Every validating node must execute both scripts together to validate the transactions. When validating an input, the node must also fetch the UTXO the input references. The node will copy the unlocking script, retrieve the referenced UTXO, and copy its locking script. If this is successful, the node will run the unlocking and locking script in sequence. The input is valid if the unlocking script satisfies the locking script conditions. Every validating node does this for every input of every transaction.

UTXOs are recorded on the blockchain, invariable, and unaffected by invalid attempts to spend them on a new transaction. Only valid transactions satisfying the unlocking conditions result in the UTXO being spent. After that, the UTXO is removed from the UTXO set.

The script is a stack-based language because it uses the stack data structure. As a refresher: a stack is a data structure with a dynamic size. It allows two operations: push and pop. Push adds an element to the top of the stack, and pop removes the topmost element. Thus, operations only act on the top of the stack and are a Last-In-First-Out, or "LIFO", queue.

The script executes programs in a left-to-right fashion. Constants, like keys, are simply

integers and are pushed onto the stack. Operators act on elements on the stack; they can push or pop elements to get parameters. If an operator has a return value, they push the result on the stack. As an example: `OP_EQUAL` pops and consumes the top two elements and pushes `1` onto the stack if they are equal and `0` when they are not.

As an example of a simple script, let us consider:

```
7 3 OP_SUB 4 OP_EQUAL
```

It is helpful to use `btcdeb` the Bitcoin script debugger to show the steps in this script. Run the following command: `btcdeb '[7 3 OP_SUB 4 OP_EQUAL]'` and the starting state is the following:

```
script   |  stack
---------+--------
7        |
3        |
OP_SUB   |
4        |
OP_EQUAL |
```

Now when the interpreter steps through the script, the following happens:

- 7 and 3 are pushed on the stack.
- The interpreter executes `OP_SUB` three, then seven is popped and subtracted, after which four is pushed onto the stack.
- 4 is pushed onto the stack.
- 4 and 4 are popped and compared. The numbers are equal; thus, 1, interpreted as `OP_TRUE` is pushed onto the stack, and the script is valid.

The final step of the script can be seen in figure 3.6. This simple arithmetic script can be turned into a locking and unlocking script. For example: take this part, `3 OP_SUB 4 OP_EQUAL` as the locking script. Then simply `7` as the unlocking would satisfy the condition. Because when the validating software combines them, it results in the previously mentioned program. The combination is insecure since anyone can spend this UTXO if they know that $4 = 7 - 3$.

Originally the unlocking and locking scripts were concatenated and then executed. The concatenation is a security vulnerability since the unlocking script can push arbitrary data on the stack and influence the locking script [Hea13]. This result meant that anyone could write a `scriptSig` that constantly evaluated to true and claim any Bitcoin. This flaw was changed in 2010, and in the current version, the scripts are executed in isolation, and the stacks are transferred between the two executions. It works as follows:

- The unlocking script is executed.
- The main stack is copied if there are no errors, like dangling operators.
- The locking script executes using the unlocking stack.
- If the result is "`TRUE`", the unlocking script has satisfied the conditions. If not, the input is invalid, and the UTXO cannot be spent.

Figure 3.7 shows a more complex script. This figure shows the previously mentioned *Pay-to-Public-Key-Hash*, abbreviated as *P2PKH*, script. The vast majority of transactions produce an output locking with this script. Essentially this locking script locks

the output to a specific Bitcoin address, a public key hash. This locked output can be unlocked by providing a public key and a signature created by the private key.

The complete script is shown in figure 3.7. When evaluating this script, it returns `TRUE` if, and only if, the unlocking script has a valid private key that matches the public key in the locking script public key hash in the locking script.

Now that the basic concepts have been introduced, more transactions with more complex conditions can be discussed. These transactions include multi-signature scripts, `Pay-to-Script-Hash`, and timelocks.

Multisignature

Multisignature scripts are scripts where k public keys are set in the locking scripts, and at least n of those need to provide a signature to unlock the UTXO. Here is k, the total number of keys, and n, the threshold required for validation. This threshold is also called the quorum. This scheme is sometimes called a `k-of-n` multi-signature scheme. A multi-signature locking script is structured like this:

```
$n$ <pubKey1> <pubKey2> $\cdots$ <pubKey$k$> $k$ CHECKMULTISIG}
```

And the unlocking script is like this:

```
0 <sig1> <sig2> $\cdots$ <sig$i$> where $n\lei\lek$
```

When this script is executed, it will only return `TRUE` if, and only if, the unlocking script has at least n signatures matching the locking script's public keys.

The keen reader might have noticed an extra zero before the signatures in the unlocking script. This zero is necessary because of a bug in the `CHECKMULTISIG` function. The function signature says it should consume $k + n + 2$ values: the k possible public keys, the n signatures necessary to reach the quorum, and finally, the two constants n and k. However, in reality, it consumes one extra element on the stack. This value is not used in the function, so a zero is commonly used. Its only purpose is as a workaround for this bug.

Pay-to-Script-Hash

While multi-signature scripts are powerful, they can become very complex to write quickly. Pay-to-Script-Hash, or P2SH, was introduced to simplify the use of complex scripts. Consider the following example to explain the usefulness of P2SH:

A shop owner uses a multi-signature script to handle all customer payments, known as "accounts receivable" or AR. Using the multi-signature scrips, all these funds are locked to require at least two signatures: one from the owner, one from one of his partners, or one from his attorney. This script offers control and protects against fraud, theft, or loss. The resulting script looks something like this:

```
2 <Owner Public Key> <Partner 1 Public Key> <Partner 2 Public Key>
    <Attorney Public Key> 4 CHECKMULTISIG
```

It is clear that this script is powerful and valuable, but simultaneously, it is cumbersome to use. The owner must convey this script to every customer when a transaction occurs. Each customer has to create a custom transaction and understand and know how to create such a script. Another problem is that this script is a lot longer because it contains four public keys, which as to be saved in every node's RAM. As previously discussed, the transaction fees are calculated per byte, and the customer will carry this extra cost.

This scenario is where P2SH is useful. Its purpose is to alleviate these difficulties and problems. Using a P2SH payment is as easy as paying to a Bitcoin address. Instead of using the complex locking script, it is replaced by the hash of the locking script. When a transaction tries to spend the UTXO, it must contain the scripts matching the hash and the unlocking script. In other words: pay to a script matching this hash; a script will be provided later when the output is spent. The hashed script is also called the redeem script because it is referenced when the UTXO is redeemed rather than locking time. Listing 3.4 shows the script without P2SH, and listing 3.5 is encoded with P2SH.

```
1 Locking Script   2 PubKey1 PubKey2 PubKey3 PubKey4 4 CHECKMULTISIG
2 Unlocking Script  0 Sig1 Sig2
```

Listing 3.4: Script without P2SH

```
1 Redeem Script   2 PubKey1 PubKey2 PubKey3 PubKey4 4 CHECKMULTISIG
2 Locking Script   HASH160 <Redeem Script Hash> EQUAL
3 Unlocking Script 0 Sig1 Sig2 <Redeem Script>
```

Listing 3.5: Script with P2SH

These listings clearly show that the locking scripts do not contain the conditions for unlocking the UTXO, only a hashed version. The unlocking script does contain the condition. This change shifts the fees and complexity of formulating the script from the sender to the transaction recipient.

When applied to the previously mentioned example, the long list of public keys, which takes up 52 bytes, gets reduced to 20 bytes. First, by applying the SHA256 algorithm followed by the RIPEMD160 algorithm. A customer can now use this much shorter script:

```
HASH160 <20-byte Hash EQUAL}
```

Which is much more concise and easier to construct. When the owner wants to spend the UTXO, he must produce the original script and the necessary signatures. The script will look something like this:

```
0 <Sig1> <Sig2> <2 PubKey1 PubKey2 PubKey3 PubKey4 4 CHECKMULTISIG>
```

The validation is performed in two stages. First, the actual redeem script is checked against its hash to make sure it is the matching script:

```
<2 PubKey1 PubKey2 PubKey3 PubKey4 4 CHECKMULTISIG> HASH160
    <Hash> EQUAL
```

If this returns `TRUE`, the unlocking script is executed:

```
0 <Sig1> <Sig2> 2 PubKey1 PubKey2 PubKey3 PubKey4 4 CHECKMULTISIG
```

A P2SH script hash can also be encoded as a Bitcoin address. Rather than providing the 20-byte hash, the store owner can provide an address to his customers to make payments. These are also Base58 encode, just like regular Bitcoin addresses. With this feature, all P2SH complexity is hidden from the person paying.

Return

Bitcoin's ledger also has uses beyond payments. People have tried to use the scripting language for other applications, such as contracts, certificates, and other services. Early attempts involved using hashes of files to prove their existence on a specific date in a transaction. This attempt was a controversial subject. Many people considered this

an abuse of the language and system. They also argued that this caused "blockchain bloat" and burdened nodes unnecessarily. These transactions also created unspendable UTXO, using the destination address as a free 20-byte field. Therefore, the UTXO set kept growing forever, and thus "bloating". Others considered this proof of how versatile the blockchain system is.

A compromise has been reached in the form of the `RETURN` operator. The compromise allows users to add 80 bytes of non-payment data to the output. In contrast with the previous method, the `RETURN` operator does provide a provably unspendable output that does not pollute the UTXO set. A `RETURN` script is very simple:

```
RETURN <data>
```

In most cases, the data field is a hash of some kind. Many applications put an identification prefix before the hash to provide an easy way to identify what data is in the data field. The previously mentioned Proof-of-Existence service has the 8-byte, ASCII encoded prefix "`DOCPROOF`".

There is no unlocking script for the "`RETURN`" script. Therefore, the output is usually zero Bitcoin because it is locked forever. If, for some reason, a "`RETURN`" script is referenced in a transaction input, the validation engine will permanently mark the transaction invalid. The execution of a "`RETURN`" operator will cause a script to always return `FALSE`. A transaction can have only one output, a "`RETURN`" output.

Timelocks

Timelocks are restrictions on transactions or transaction outputs, allowing them to be spent only after a point in time. Bitcoin has several timelock implementations. Initially, it had a transaction-level timelock implementation, but this quickly proved too coarse and unwieldy. Thus, two more features were implemented, which offer UTXO-level timelocks.

First, the transaction-level timelock, known as `nLocktime`, was implemented. In most transactions, this is set to zero to propagate and execute the transaction immediately. This `nLocktime` is non-zero and lower than 500 million; it is interpreted as block height[1]. This interpretation means the transaction is invalid if included before the specified block height. When the `nLocktime` value is higher than 500 million, it is interpreted as a Unix Epoch timestamp, and the transaction is not valid before this timestamp. A transaction with a future `nLocktime` must be kept in the mempool by the originating node and only be transmitted to the network after they are valid. When it gets transmitted before the specified time, it gets rejected by the first nodes and is not propagated.

This transaction-level lock time does have significant limitations. While the transaction can be spent in the future, it is not impossible until then. The formulation might be confusing and is easier to explain with an example:

Alice signs a transaction that sends output to Bob's address and the `nLocktime` to a month in the future. She sends this transaction to Bob to hold on. The transaction implies that Bob can only transmit the transaction once a month has passed, and Bob may send the transaction after a month. There are several caveats. Alice can create a new transaction referencing the same inputs without a timelock. Thus double spending the UTXO before the months have elapsed. Bob has no guarantee that Alice will not do this. The only guarantee using this scheme is that Bob can redeem the transaction once the month has passed. For Bob to have a guarantee that he can redeem his funds, there must be a restriction on the UTXO itself so that every node

[1]The block height is the number of blocks preceding the current block.

can verify the timelock. The following type of timelock implements this: Check Lock Time Verify (CLTV).

CLTV is a per-output timelock rather than a transaction-level timelock. More straightforward: adding the CLTV opcode to a redeem script restricts the output to be only spendable after a specified time has elapsed. The opcode takes a single input argument, a number in the same format as the `nLocktime`. The CLTV halts the verification if it evaluates to `FALSE`, as the verify suffix implies.

Locking an output with CLTV is relatively simple. Insert it into the redeem script of the output of a transaction. Take the previous example, Alice wants Bob to be able to redeem his funds a month from now:

```
<now + 1 month> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <PubKeyBob>
    EQUALVERIFY CHECKSIG
```

In this script, the `<now + 1 month>` is either the block height, approximately the current block height + 12960 blocks because each block takes approximately ten minutes to create, or the time value, current Unix time + 7760000 seconds. The `DROP` opcode is necessary because CLTV keeps the time parameter on the stack. For Bob to spend the UTXO, he does the following: he creates a transaction that references the UTXO as an input. He uses his signature and public key in the unlocking script as usual. Finally, he sets the `nLocktime` of the transaction to equal or greater than the timelock set in the CLTV by Alice. Then Bob broadcasts the transaction. If the `nLocktime` is greater or equal to the CLTV parameter, the script execution continues as usual; if not, the execution is halted, and the transaction is marked invalid.

CLTV and `nLocktime` are absolute timelocks; they specify an absolute point in time. The following paragraphs discuss two relative timelocks. Which means they specify an elapsed time concerning the confirmation of the output. Do note that the timer starts once the transaction is confirmed. The delayed timer allows for holding certain interdependent transactions off-chain until a specific time has elapsed since a transaction has been confirmed.

Like absolute timelocks, there is a transaction-level feature and an output-level opcode. The transaction-level timelock is implemented as a consensus rule as a transaction field value `nSequence`. Originally it was intended to modify transactions while they were in the mempool, but this was never implemented. In this use-case, a transaction containing a `nSequence` field with a value lower than $2^{32} - 1$, or `0xFFFFFFFF`, meant that the transaction was not yet finalized. This transaction would then be kept in the mempool until it was substituted by a transaction spending the same UTXO with a lower `nSequence` value. Once the node receives a transaction with `nSequence` value `0xFFFFFFFF`, the transaction is considered finalized and can be mined.

When a transaction does not use timelocks, the `nSequence` value is usually set to `0xFFFFFFFF`. If a transaction wants to utilize the absolute timelocks, the `nSequence` value is usually set to a value lower than `0xFFFFFFFF`; most of the time, the value is set to `0xFFFFFFFE`.

Today, `nSequence` creates a consensus-enforced relative timelock. If the most significant bit, `bit 1«31`, is not set, consensus rules for relative timelocks apply. If this bit is set, the `nSequence` field is used for other purposes, such as `nLocktime`, CLTV, and further developments.

When the most significant bit is not set, the transaction is only valid if it has aged by the relative timelock amount. For example, if the `nSequence` value of a referenced UTXO is set to 10, the transaction is only valid when ten blocks have elapsed since the UTXO has been mined. When a transaction can reference multiple time-locked inputs, they all must satisfy the age requirement for the transaction to be valid.

The nSequence value is specified in blocks or seconds but differs from nLocktime. Unlike nLocktime, it contains a type-flag at the 23^{rd} bit to differentiate between blocks and seconds. If the type-flag is set, the nSequence value is interpreted as a multiple of 512 seconds. If it is not set, the value is interpreted as blocks. The value is defined as bits 0 to 16. Once the disable and type-flag are evaluated, the nSequence field is masked with a 16-bit mask. Figure 3.8 shows the nSequence bit-field.

Like absolute timelocks, there is a script opcode alternative to nSequence. The opcode is called CHECKSEQUENCEVERIFY, abbreviated as CSV.

When a UTXO script using CSV is evaluated, it can only be spent when the nSequence value is greater than the parameter provided to the CSV opcode. More simply, CSV restricts spending a UTXO until a certain amount of blocks or seconds has passed relative to when the UTXO was mined. The CSV parameter must also match the format of the nSequence value. When the nSequence value specifies blocks, so must the CSV value.

When relative timelocks were introduced, there was also a change in time calculation for timelocks. Bitcoin is a decentralized network, and every node has its time perspective. Network latency exists, which means that only some events coincide everywhere. The network must account for this latency. Eventually, everything will be synchronized every ten minutes when a consensus is reached, and the network will have a shared ledger.

Different nodes have different clock accuracies. Thus, the consensus protocol needs to give some leeway to nodes to miners. The miners need to set the timestamp in the block headers after all. There is a catch; however, in some circumstances, it is in a miner's best interest to lie about a timestamp to earn extra fees by including time-locked transactions.

A new consensus measurement has been introduced to remove this incentive: the *Time-Median-Pass*. This measurement is calculated by taking the timestamps of the past 11 blocks and calculating the median. The median time is then used as consensus time and used for all timelock calculations. The median time of 11 blocks equates to roughly one hour. Thus, one hour behind wall time is taken as the current time. The median time reduces the influence of any one timestamp, and no miner can manipulate the timestamp to extract more transaction fees from time-locked transactions. This time is then used for nLocktime, CLTV, nSequence, and CSV calculations.

Flow control

The final topic is flow control in Script. These are familiar as most programming languages have these constructs, but these look different in Script. At a fundamental level, these opcodes allow a user to construct an unlocking script that can be unlocked in several ways.

The conditional expressions can have an infinite nesting depth and, consequently, be very complex with many execution paths. There is one limited factor. However, consensus rules put a limit on the size of a script.

The script implements the following flow control opcodes: IF, ELSE, ENDIF, and NOTIF. They can also include boolean operators: BOOLAND, BOOLOR, and NOT. Because Script is a stack-based language, these look backward when used in practice. In procedural languages, a classic if-statement looks something like this:

```
1 if(condition == TRUE) {
2     // Code in true path
3 } else {
4     // Code in false path
```

```
5 }
6 // Code which always runs
```

Listing 3.6: Flow control in a procedural language

In Script, the condition comes before the actual if-statement:

```
1 condition
2 IF
3     // Code in true path
4 ELSE
5     // Code in false path
6 ENDIF
7 // Code which always runs
```

Listing 3.7: Flow control in Script

Another type of flow control is `VERIFY` opcodes. Two were discussed earlier regarding timelocks. When these opcodes evaluate to `FALSE`, the execution of the script is terminated, and the transaction is marked as invalid. Terminating means no alternative paths; they act as a guard clause or exception. Another example can be found in the *Pay-to-Public-Key-Hash* script:

```
OP\_DUP OP\_HASH160 <pubKeyHash> OP\_EQUALVERIFY <pubKey> OP\_CHECKSIG
```

Here the receiver needs to provide his public key and signature: `<sig> <pubKey>`. If the hashed public key does not equal the expected hash, the execution will be terminated when the `OP_EQUALVERIFY` is evaluated and the top two elements on the stack do not match. The locking script can be rewritten using if-statements, while the unlocking script stays the same. Such a locking script can be seen in listing 3.8; the `VERIFY` construction is much more efficient since it requires fewer opcodes than the alternative. If-statements are only helpful if more than one execution path is required. Always use `VERIFY` opcodes when a guard clause is required.

```
1 OP_HASH160 <pubKeyHash> OP_EQUAL
2 IF
3     <pubKey> OP_CHECKSIG
4 ENDIF
```

Listing 3.8: `EQUALVERIFY` with if-statements

Another example is when there are two signers, and either one can redeem the UTXO. In the previously mentioned multisig construct, this would be expressed as a 1-of-2 multisig script. This script is rewritten using if-statements for demonstrating the executions path in listing 3.9 There is no condition since the redeemer will decide who redeems the UTXO. The unlocking script will look like this: `<sig1> 1` for the first execution, or `<sig2> 0` for the second execution path.

```
1 IF
2     <pubKey1> OP_CHECKSIG
3 ELSE
4     <pubKey2> OP_CHECKSIG
5 ENDIF
```

Listing 3.9: Multisig with if-statements

If more than two execution paths are necessary, if-statements must be nested. The if-statements create a maze of paths for unlocking scripts to map the desired path. Listing 3.10 shows three paths. The unlocking script has to provide `TRUE` and `FALSE` values in the correct order in order to arrive at the correct path. The unlocking script needs to end in `1 0` to get to path 2. The outer if pops the zero and thus executes

the first ELSE path. Then the inner if pops the one and executes the inner IF path, which selects path 2. Users can create complex unlocking scripts with hundreds of paths using these constructions.

```
1  IF
2      // Path 1
3  ELSE
4      IF
5          // Path 2
6      ELSE
7          // Path 3
8      ENDIF
9  ENDIF
```

Listing 3.10: "Maze" of execution paths

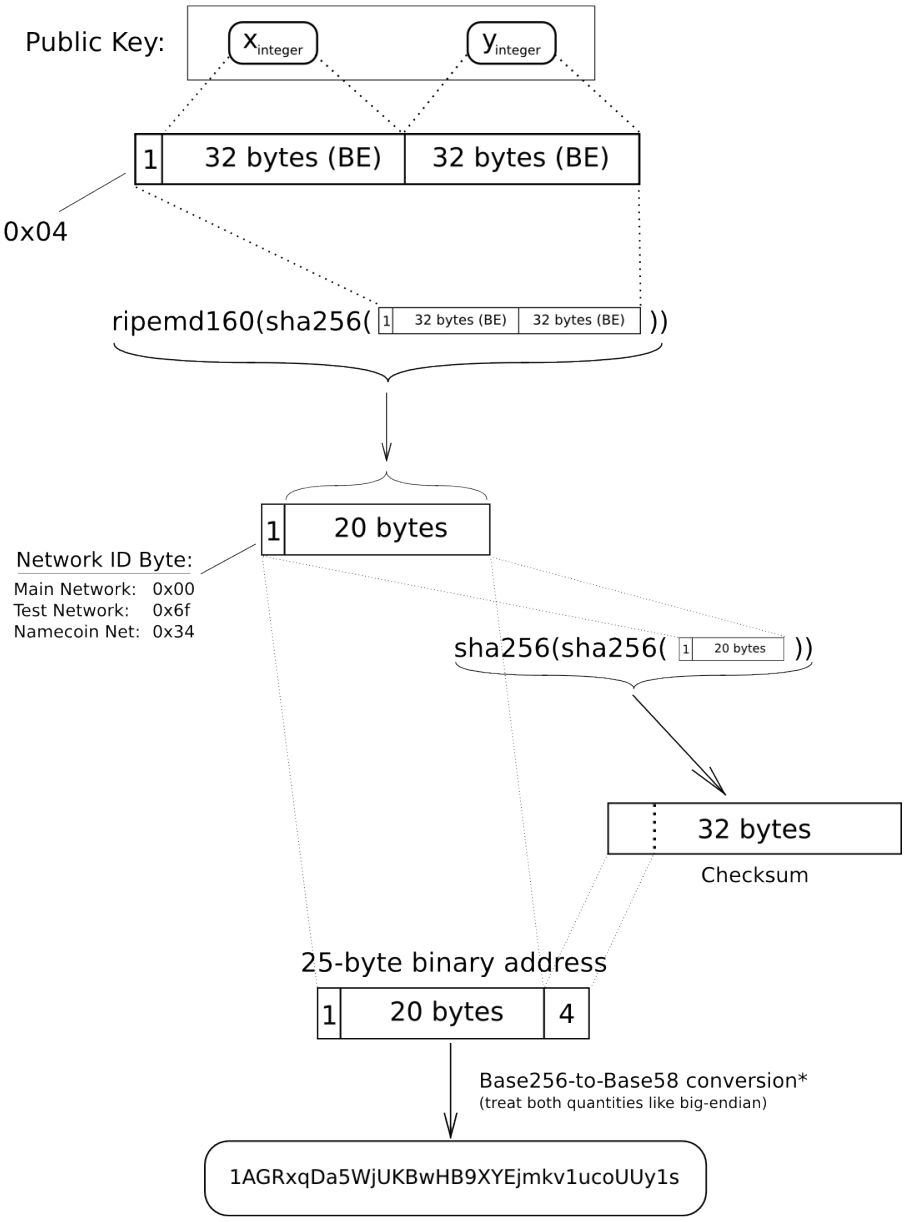
A complex script will combine the previously discussed concepts to conclude this section. A store owner and two partners, together with their lawyer, run a company. They use a majority rule setup. The majority rule means that two out of three can make decisions. However, when there are problems with keys, they want the lawyer to retrieve funds with one partner after 30 days. Also, when all three partners are unavailable, the lawyer can retrieve funds after 90 days.

The script can be seen in listing 3.11. There are three paths:

- The first path is a simple 2-of-3 multisig construct for the three partners. The quorum gets set on line three, and line nine executes the multisig. The path can be reached by putting TRUE TRUE at the end of the unlocking script.
- The second path can be reached 30 days after creating the UTXO. This path requires the lawyer's signature and a single partner's signature. The quorum is set at line seven. The path can be reached by putting FALSE TRUE at the end of the unlocking script.
- The final path can be reached after 90 days and only requires the lawyer's signature. The path requires a single FALSE at the end of the unlocking script.

```
1  IF
2      IF
3          2
4      ELSE
5          <30 days> CHECKSEQUENCEVERIFY DROP
6          <lawyerPubkey> CHECKSIGVERIFY
7          1
8      ENDIF
9      <pubKey1> <pubKey2> <pubKey> 3 CHECKMULTISIG
10 ELSE
11     <90 days> CHECKSEQUENCEVERIFY DROP
12     <lawyerPubkey> CHECKSIG
13 ENDIF
```

Listing 3.11: A more complex script



*In a standard base conversion, the 0x00 byte on the left would be irrelevant (like writing '052' instead of just '52'), but in the BTC network the left-most zero chars are carried through the conversion. So for every 0x00 byte on the left end of the binary address, we will attach one '1' character to the Base58 address. This is why main-network addresses all start with '1'

Figure 3.2: Generation of a Bitcoin address from a public key.

Figure 3.3: Relationship between private key, public key, and Bitcoin address.

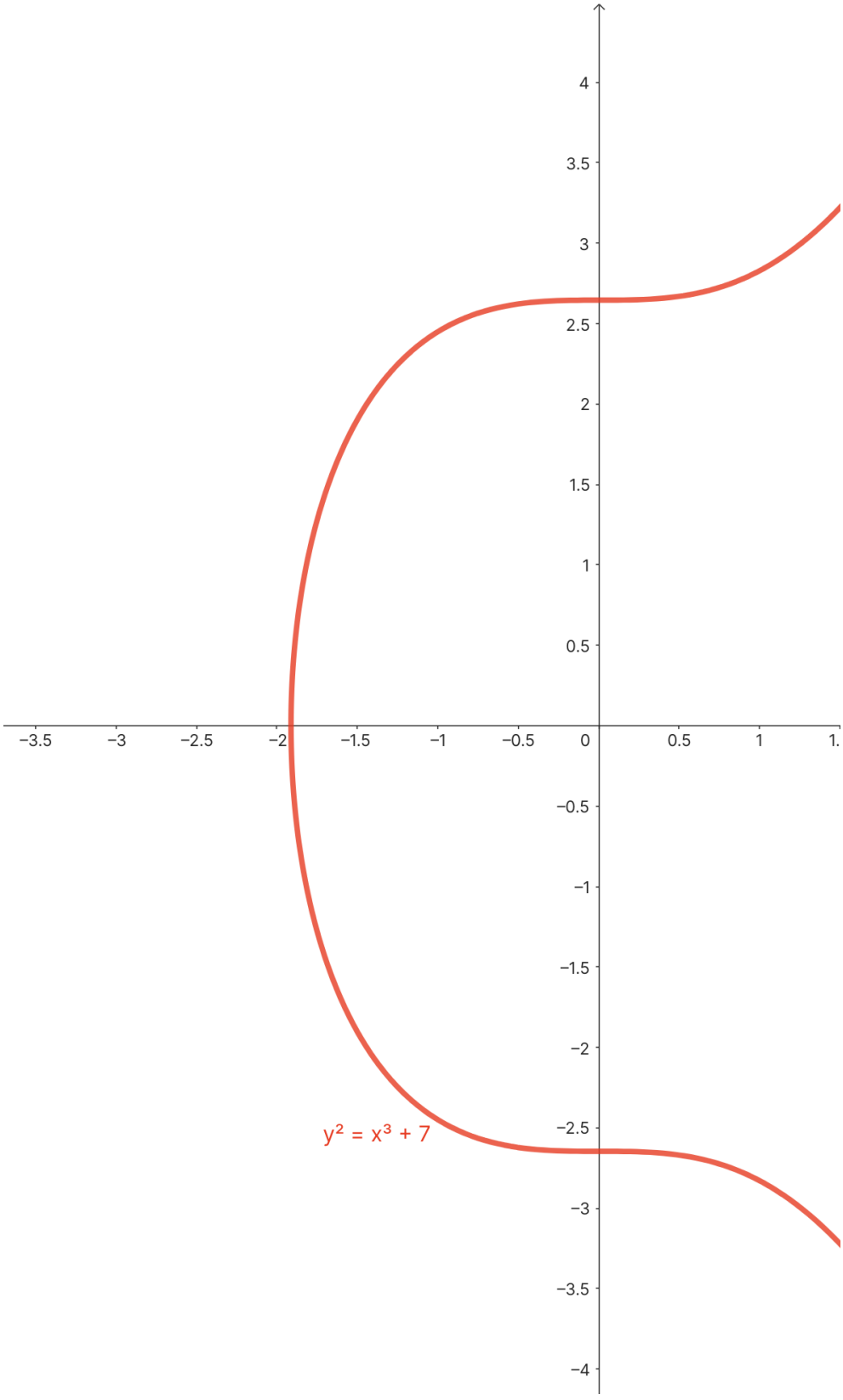


Figure 3.4: The elliptic curve used by Bitcoin.

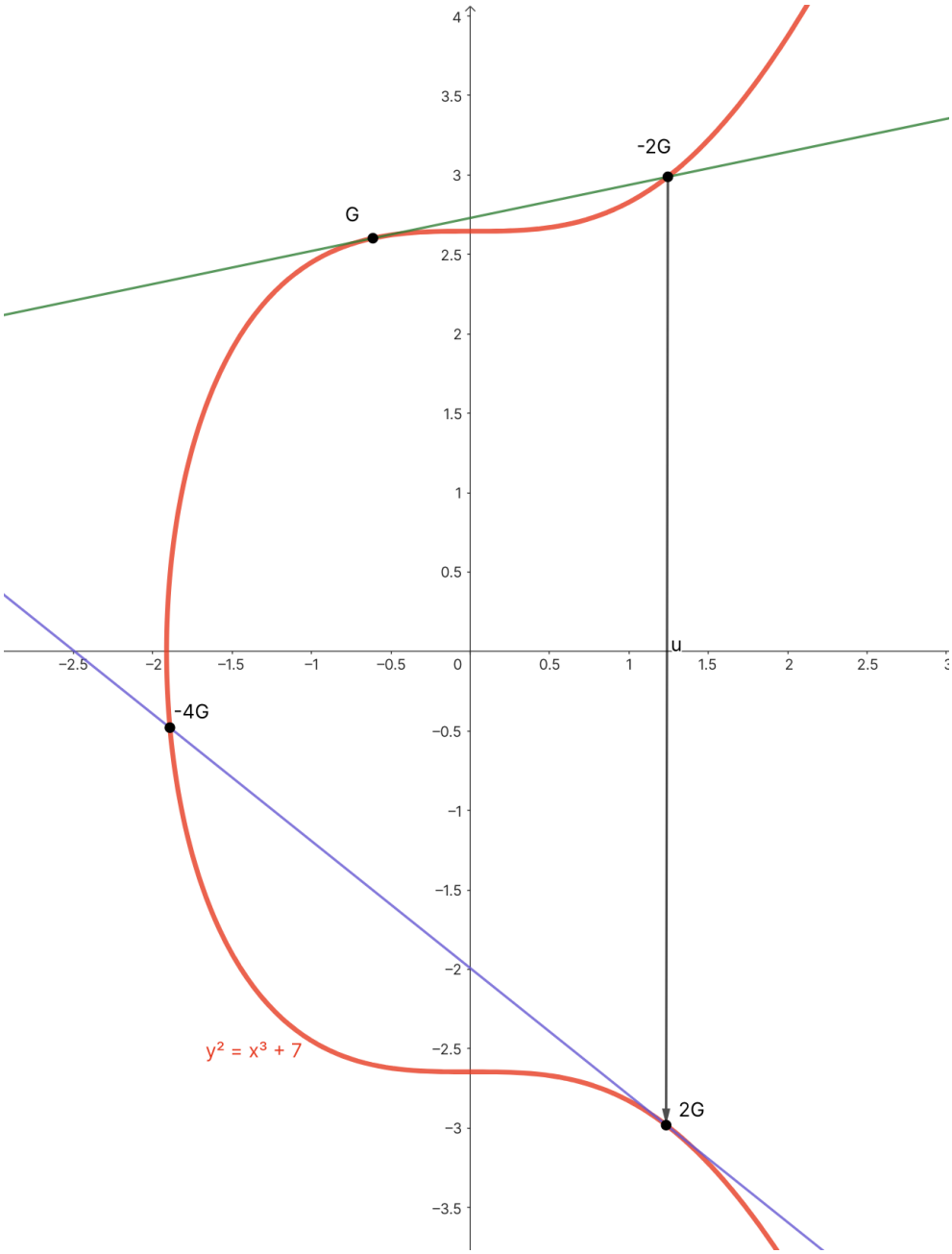


Figure 3.5: How public keys are calculated on an elliptic curve.

```
script   |  stack
---------+--------
OP_EQUAL |      04
         |      04
#0004 OP_EQUAL
btcdeb> step
                 <> POP  stack
                 <> POP  stack
                 <> PUSH stack 01
script   |  stack
---------+--------
         |      01
```

Figure 3.6: OP_EQUAL in a Bitcoin script.

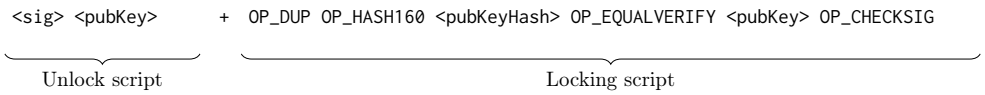


Figure 3.7: The common *Pay-to-Public-Key-Hash* script.

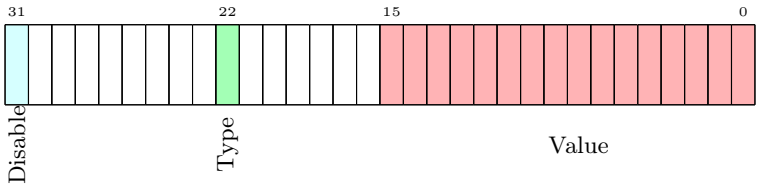


Figure 3.8: The nSequence encoding

Chapter 4

Ethereum

Ethereum is often called a "world computer", but what does this description mean? Ethereum is an open-source, globally decentralized computing infrastructure. This computer executes smart contracts and uses a blockchain to synchronize and store system state changes. Its native token, called ETH, is used to measure and constrain the costs of executing smart contracts. The smallest denomination for ETH is called a wei and is defined as: 1 wei $= 1 \cdot 10^{-18}$ether. This combination allows users to develop decentralized applications with high availability, auditability, and transparency. It is important to note that almost every permissionless blockchain has a native token. These are necessary to incentivize users to keep the system running and to pay for using system resources.

Some people considered Bitcoin to be "blind, deaf, and dumb". People meant that users could not do more than push Bitcoin around. Users cannot issue their currencies or write applications or contracts.

Ethereum does share many elements with Bitcoin. Bitcoin and Ethereum both use a peer-to-peer network to connect nodes, a consensus algorithm to synchronize state updates, and use cryptographic primitives.

However, the purpose of Ethereum is also very different from Bitcoin. Ethereum's purpose is not primarily a digital currency payment network. Its currency is necessary for running Ethereum but is intended as a utility currency to use the Ethereum system.

Unlike Bitcoin, which has a very primitive scripting language, Ethereum is a general-purpose programmable blockchain. It runs a virtual machine called the *Ethereum Virtual Machine or EVM*, which can run programs of arbitrary complexity. Where Bitcoin's programming language is intentionally limited to simply evaluating true/false spending conditions, Ethereum's programming language, called *Solidity*, is Turing complete.

However, Turing completeness is a double-edged sword, especially in an open-access system like Ethereum. For example, modern printers are sometimes Turing complete; one can send files to freeze the printer. The flexibility of a Turing complete system can bring security and resource management issues with it.

It is only possible to predict a specific program's path by running it. In Ethereum, this can pose a problem as nodes are required to validate every transaction, which means running them. Moreover, given that Ethereum cannot predict whether a smart contract will stop, the node runs the risk of running forever. Something is needed which would constrain computing resource usage. Ethereum's answer is a metering mechanism

called *gas*. As the EVM executes a smart contract, it counts every instruction, and each instruction has a predetermined cost in gas. A transaction has to include an amount of gas, the upper limit of what can be consumed if it intends to run a smart contract. The smart contract will immediately be stopped when this available limit is exceeded.

A user can obtain gas by buying it with ether. If a user wants to run a smart contract, he has to include ETH designated for buying gas with the transaction. The gas price fluctuates, and its price is proportional to the traffic on the network. Any gas not used by the contract execution is refunded to the user who sent the transaction. In short: gas is purchased for the transaction, it is computed, and any unused gas is refunded.

Ethereum was introduced when people recognized the utility and power of the Bitcoin model. People were also trying to move past mere cryptocurrency applications. When attempting to do so, developers had to decide: build on top of Bitcoin or create something new. Building on top of Bitcoin meant inheriting its limitations, intentional or not. These constraints in transaction types, data types, and data storage limited the type of application which could be run on top of Bitcoin. When developers wanted to extend Bitcoin, this meant it had to be run off-chain in a layer on top of Bitcoin. An additional layer means sacrificing many advantages of a public blockchain. Thus, for projects which require more versatility while staying on-chain, this means creating a new project and a new blockchain.

Two examples of projects using a layered approach are Mastercoin and Colored Coins. Mastercoin offers rudimentary smart contracts on Bitcoin, and Colored Coins mark specific Bitcoin to mean ownership of a real-world asset.

The introduction of Ethereum was similar to when JavaScript was introduced to the web. Before JavaScript, the web browser worked fine, but websites were static. JavaScript allowed websites like YouTube and Google to offer dynamic content. The evolution of cryptocurrencies is analogous; with Bitcoin, users can send, receive, and display transactions. Users can use metadata in transactions to do exciting things, but actual distributed applications or custom tokens are not feasible.

Bitcoin tracks the state of Bitcoin and its ownership of them. One can think of this as a distributed state machine. Every transaction causes a state transition that is constrained by consensus rules.

Ethereum is also a state machine but tracks the state of general-purpose data. Ethereum has memory that can hold data and programs and uses the Ethereum blockchain to track how this memory changes over time. Like a traditional computer, Ethereum can load and execute these programs and store the resulting state changes of the programs on its blockchain. Two differences exist between the Ethereum model and traditional computers: the state is distributed, and consensus rules constrain the state changes. Thus, Ethereum is a "world computer" because it is distributed globally and operates under consensus.

4.1 Nodes

The Ethereum client is, again, like Bitcoin, just an application that implements the Ethereum specification. The client communicates with other clients over a peer-to-peer network. The specification ensures that different clients can work together since it standardizes the protocol. Different implementations of an Ethereum client can be written in different languages by different teams, but as long as they comply with the reference specification, they will work together.

This open specification contrasts with Bitcoin, where there is no formal specification that teams can read and implement. In Bitcoin, there is only a reference implementation, the Bitcoin Core. Ethereum's specification is called the *Yellow Paper* and can be read here: [Woo22].

Many independent but interoperable Ethereum clients exist due to the open specification. This diversity of clients has some advantages. If one implementation has a security vulnerability because of an exploit, its developers can scramble and patch it. While the exploit is being fixed, other implementations of clients can keep the network running.

The network relies on independent and dispersed nodes for its operation. Each node helps new nodes obtain data via the peer-to-peer network and verify transactions and contracts. As of writing, a full node downloads 1023.54 GB of data to store on a local drive. This data increases over time as new transactions and blocks are added.

Luckily a full *mainnet* node is unnecessary for every application. The mainnet is the main public network where all transactions with real ETH and value occur. Developing applications for Ethereum does not require a full mainnet node. All a developer needs is a *testnet* node or local private network. A testnet is a public blockchain network primarily used by developers to test protocol upgrades and smart contracts before deploying them to the main net. In this thesis development of Ethereum applications was completed using a local private network with *Foundry*. The development of Ethereum applications is discussed more in-depth in section 4.7.

Another option is remote clients. These do not store a copy of the blockchain locally or even validate blocks and transactions. Remote clients can create and broadcast transactions and connect to existing networks. Interacting with these clients is generally easy because they provide an API.

4.2 Accounts

Where Bitcoin uses the UTXO model for handling transactions, Ethereum uses an account model.

There are two types of accounts; *externally owned accounts (EOA)* and *contract accounts*. Externally owned accounts are accounts that possess a private key. Owning a private key means control over access to funds and contracts. Contract accounts have a smart contract code, something externally owned accounts do not have. Contract accounts also have no private key, which means a simple key does not control them, but by the logic of its smart contract code. Contracts do have addresses, just like EOAs. Contracts having addresses means contracts can also send and receive funds. A key difference is that when a transaction is sent to a contract, the contract runs on the EVM. The contract is run using the transaction's data as input. The transaction's data includes which function in the contract to call and which arguments to pass to the function.

Because a contract does not possess a private, it cannot initiate sending a transaction. Only users, EOAs, can do this. Contracts react to transactions by calling other contracts. Contracts reacting to each other can lead to complex execution paths. How to build and program smart contracts will be discussed in section 4.7.

4.3 Keys and Addresses

This subsection will discuss the *public key cryptography*, PKC, used in Ethereum. Like Bitcoin, public key encryption is used to determine ownership of funds.

As mentioned in the previous subsection, Ethereum has two kinds of addresses: EOA en contract accounts. Ownership of funds by EOAs is determined through private keys, addresses, and signatures. Account addresses are derived from private keys, meaning a private key determines an Ethereum address known as an account.

Like in Bitcoin, private keys are never used directly. They are never transmitted or stored in Ethereum. Only account addresses and signatures are ever transmitted.

Access and control of funds in an account are exerted through digital signatures. Every transaction requires a valid signature to be included in a block. Anyone with access to the private key of an account has control over any ETH the control possesses.

In the payment portion of a transaction, the receiver of funds is represented by an address. This model is analogous to the beneficiary in a bank transfer. The address of an EOA is derived from the public key, but private-public key pairs do not represent contracts.

Ethereum, again like Bitcoin, also uses elliptic curve cryptography for key generation. It uses a different hashing algorithm to achieve a 256-bit number, namely the Keccak-256 hashing algorithm. Keccak-256 belongs to the SHA-3 family of hashing functions. Keccak-256 is more robust than SHA-256, which is used in Bitcoin.

Public keys are derived the same way as Bitcoin public keys, as two points on an elliptic curve joined together. These two numbers are produced from a one-way calculation with the private key. The elliptic curve is the same one used by Bitcoin: `secp256k`.

Ethereum addresses are generated in the following way:

- Start with a private key and use elliptic curve multiplication to derive a public key.
- Use the Keccak-256 function to calculate the public key hash.
- Key the last 20 bytes of the hash; this is the Ethereum address.

Ethereum addresses lacked a checksum until 2016. Users could send funds to invalid addresses and were essentially losing funds. EIP-55 was proposed in 2016. An EIP is an Ethereum Improvement Proposal meant to improve and advance Ethereum continually. EIP-55 enhances address validation to prevent transaction errors that can occur from mistyping.

Before EIP-55, Ethereum addresses were simple hexadecimal representations, as described above, prone to errors from typos that could lead to misdirected transactions. To solve this, EIP-55 introduced a checksum implemented via case sensitivity. It works as follows:

- The Ethereum address is hashed again using the Keccak-256 hash function.
- Each character in the original address is compared with the corresponding character in the hash.
- If the hash character is `0x08` or higher, the corresponding address character is capitalized. The character remains lowercase if it is `0x07` or lower.

This approach effectively implements case sensitivity into the address, creating a checksum. If any character is in the wrong case, the checksum fails, and the transaction is halted. A key benefit of EIP-55 is its backward compatibility. Older addresses can still be accepted, which can be adopted incrementally. However, EIP-55 is not mandatory, and its implementation varies from client to client. Despite this, many

have adopted it due to its efficacy in catching typing errors that can lead to incorrect transactions.

4.4 Transactions

Ethereum transactions serve as the backbone of the Ethereum network, facilitating interactions between accounts. Understanding the architecture and lifecycle of Ethereum transactions is critical to understanding how Ethereum operates as a distributed computing platform. Transactions are signed messages created by EOAs, sent through the Ethereum network, and recorded in blocks on the blockchain. Transactions also trigger a change of state and can cause contracts to trigger.

Each transaction follows a strict structure. This structure is necessary because each transaction is serialized before being transmitted over the Ethereum network. Since each client has a different implementation, once a client receives a serialized transaction, it will store the transaction in a custom data structure, thus each differently in memory. The only standard format is the transaction format over the network.

A transaction contains the following data:

- Nonce: This is a value set by the sender, an EOA, of the transaction. It is used to prevent double-spending and to order transactions. For every new transaction, the nonce value increases by one.
- To: This is the recipient address of the transaction. It can be another user's account or a smart contract's address.
- Value: This is the amount of ETH transferred from the sender to the recipient.
- Gas Limit: This represents the maximum amount of computational work, measured in "gas", the sender is willing to allocate for the transaction or contract execution.
- Gas Price: This is the price the sender is willing to pay for each gas unit. This price is usually denominated in Gwei, where 1 Gwei is one billionth of an Ether.
- Data: This field is optional and usually carries extra data or the encoded function call to a smart contract.
- v, r, s: These are cryptographic pieces of data that are part of the digital signature of the transaction. They prove that the transaction was indeed created by the owner of the sender's address.

Note that it does not contain a "from" field. The public key can be derived from the v, r, and s fields, and the address can be derived with the public key.

Consider the following raw transaction:
`0xf86d82025f8502540be40083030d4094f0109fc8df28`
`3027b6285cc889f5aa624eac1f55843b9aca008018080`
This string is a hexadecimal representation of a raw Ethereum transaction, encoded using Recursive Length Prefix (RLP) encoding. RLP is a space-efficient binary serialization scheme used throughout Ethereum. Each of the transaction's fields is RLP-encoded.

RLP encoding is a method used in Ethereum to serialize and deserialize structured binary data. Its primary purpose is to encode nested arrays of binary data of arbitrary length, making it integral to storing data structures efficiently.

The rules for RLP encoding are as follows:

- For a single byte whose value is in the [0x00, 0x7f] range, that byte is its own RLP encoding.
- For a binary string of length 0-55 bytes, the RLP encoding consists of a single byte with value 0x80 plus the length of the string, followed by the string. The range of the first byte is thus [0x80, 0xb7]
- For a binary string of length more than 55 bytes, the RLP encoding consists of the string length in binary form, prefixed by a single byte with value 0xb7 plus the length in bytes of the string length. Then follows the string.
- For a list of concatenated RLP encodings, if the total payload of all the items forming the list is 0-55 bytes long, the RLP encoding consists of a single byte with value 0xc0 plus the length of the concatenated encoding, followed by the concatenated encoding.
- If the total payload of a list of concatenated RLP encodings is more than 55 bytes long, the RLP encoding consists of the total payload length in binary form, prefixed by a single byte with value 0xf7 plus the number of bytes needed to encode the total payload length, followed by the concatenated encoding.

Here is a breakdown of the example:

- f86d: This is the total byte length of the transaction.
- 82025f: This is the nonce. It shows that this is the 150th transaction by the sender (025f in hexadecimal is 150 in decimal).
- 8502540be400: This is the gas price, which is 10 Gwei in this example (02540be400 in hexadecimal is 10000000000 in Wei).
- 83030d40: This is the gas limit, set at 200000 in this example (030d40 in hexadecimal).
- 94f0109fc8df283027b6285cc889f5aa624eac1f55: This is the recipient's address.
- 843b9aca00: This is the value of ETH being sent (3b9aca00 in hexadecimal is 1 ETH in Wei).
- 80: This represents an empty data field.
- 80: Represents v, which is 27 in this case.
- 80: Represents r, which is empty in this case.
- 80: Represents s, which is empty in this case.

Note that the raw transaction above is not signed. When signing a transaction, the values of v, r, and s are replaced with the respective parts of the ECDSA signature.

After a transaction is signed, it can be broadcast to the Ethereum network for miners to pick it up and include it in a block. Any changes to the transaction after it is signed would require it to be re-signed, as the signature would no longer be valid.

The nonce is a counter that ensures each transaction is processed only once and prevents replay attacks. It is a crucial part of the Ethereum transaction structure for maintaining the integrity and order of transactions.

Every EOA has a nonce associated with it, set to zero at the time of account creation. For every outgoing transaction, an account makes, its nonce increases by one.

When a transaction is issued, it includes the nonce value in its structure. Miners and validators use this nonce to determine how transactions should be processed. For instance, a transaction with a nonce of 1 will not be processed until the transaction with a nonce of 0 has been processed.

This mechanism ensures two things:

The uniqueness of transactions, since each transaction from an account must have a unique nonce, this prevents replay attacks where a malicious actor might try to re-broadcast a transaction to make it appear that the original sender issued the same transaction multiple times.

Consider the following example:

Assume that Alice has an account on the Ethereum network. When she first creates the account, the nonce for her account is set to 0.

Alice then decides to send 1 ETH to Bob. She creates a transaction, includes the nonce of 0 in the transaction data, signs it, and broadcasts it to the network. Miners pick up the transaction, validate it (which includes checking that the nonce is correct), and then include it in a block. Once the transaction is included in a block and the block is added to the blockchain, Alice's account nonce increases to 1.

Suppose a malicious actor, Eve, tries to replay Alice's transaction to force Alice to send another 1 ETH to Bob. Eve rebroadcasts the same transaction that Alice broadcasted earlier. However, since the transaction includes a nonce of 0, and Alice's current nonce is now 1, the Ethereum nodes reject this transaction as invalid. The transaction is rejected because they know Alice has already completed the transaction with a nonce of 0, and they are now expecting a transaction from Alice with a nonce of 1.

Transaction order, transactions from an account must be processed in the order determined by their nonce. The order is crucial when multiple transactions from the same account are issued rapidly or when network delays might otherwise result in transactions being received out of order.

Transaction order is illustrated in the following example:

Suppose Alice has a current nonce of 5 for her account. She wishes to send two transactions:

1. A transaction to send 2 ETH to Bob.
2. A transaction to send 3 ETH to Charlie.

Alice first constructs the transaction for Bob and includes the nonce value of 5. She signs this transaction and broadcasts it to the Ethereum network. However, the transaction propagates slowly to the miners due to network congestion or other issues.

Next, Alice constructs the transaction for Charlie, including the nonce value 6. She signs this transaction and broadcasts it to the Ethereum network. This transaction propagates quickly and reaches the miners before the transaction is intended for Bob.

Under normal circumstances, the transaction for Charlie should be processed first because it reached the miners earlier. However, due to Ethereum's nonce-based ordering rule, the miners will only process Alice's transaction for Charlie once they have processed her transaction for Bob.

The miners know that they missed a transaction from Alice because they see the nonce of 6 in Charlie's transaction and realize they still need to process the transaction from Alice with a nonce of 5.

While the nonce is a simple integer counter, it plays a crucial role in maintaining the security and consistency of transactions on the Ethereum network. This counter contrasts with Bitcoin's UTXO model, where such a counter is optional due to the nature of chaining transactions together.

It is also important to note that including a nonce makes concurrency difficult in Ethereum. Consider that there are independent applications that are generating transactions from the same address. How would multiple applications coordinate generating, signing, and broadcasting transactions? The selection of nonces would be a severe problem. A solution would be to have one application responsible for assigning nonce on a first-come-first-serve basis.

In the Ethereum network, computation and storage are not free. Instead, they are metered using a unit called *gas*. Every operation that the Ethereum Virtual Machine executes, including computations and memory storage, requires a certain amount of gas.

Every transaction requires computational resources to execute, which are not infinite. To measure and limit the computational work needed, Ethereum introduces the concept of gas. Each operation, like addition, subtraction, or more complex data manipulation, has a specified gas cost.

The *gas limit* is a value set in each transaction, representing the maximum amount of gas a transaction can consume. The gas limit protects users from incorrect code or errors consuming all their ETH due to infinite loops or excessive computational operations.

While gas measures the computational work, the gas price is the amount of ETH a sender is willing to pay each gas unit. The gas price is typically denoted in gwei, where 1 ETH equals 1,000,000,000 (one billion) gwei.

The transaction's sender sets the gas price, incentivizing miners to prioritize their transactions. Miners are free to choose which transactions to include in the blocks they mine, and they are incentivized to choose transactions that offer a higher gas price. This choice leads to a market for block space, where users can compete on gas prices to have their transactions processed more quickly.

The total transaction fee that a sender must pay is the product of gas used and the gas price, transaction fee = gas used $\cdot$ gas price. If a transaction does not use all the gas corresponding to the gas limit, the excess is refunded to the sender. However, if the gas used by a transaction reaches the gas limit, the transaction fails and halts immediately, but the fee is still paid.

This design offers a balance, enabling Ethereum to run computations and store data while at the same time incentivizing efficiency and limiting resource usage. Too high a gas price might result in quick transaction processing but at a high cost. On the other hand, a low gas price can result in the transaction not being processed at all if miners favor transactions with higher gas prices.

By understanding the relationship between gas, gas limit, and gas price, Ethereum users can optimize their transaction costs based on network demand while processing their transactions promptly.

The value and data are the primary payloads of a transaction. A transaction with only a value and no data is a payment, only data and no value an invocation, and with both, it is both a payment and invocation.

Payments behave differently depending on what kind of address they are sent. For EOAs, Ethereum will have to perform a state change. If this address is unknown to Ethereum, it must be initialized. Its balance will be the amount of ETH in the transaction.

When the destination is a contract, the EVM will have to execute the contract. It will attempt to execute the function called in the data field. If the transaction has an empty data field, the EVM will attempt to call the fallback function. If the fallback function

is payable, the EVM calls the function. Finally, if it is not payable, the transaction will increase the balance of the contract like an EOA.

A contract can also reject incoming payments by throwing an exception. A state change also happens for contracts if the function called by the transactions terminates successfully.

When a transaction does contain data, it will be interpreted as a contract invocation. This payload is a serialized encoding of the following:

- Function selector: The first four bytes of the Keccak-256 hash of the function prototype. The selector identifies which function the transaction wants to invoke. The function prototype is the function name followed by the data type of each argument. The data types are enclosed in parentheses and separated by commas.
- Function arguments: The encoded arguments for the invoked function.

Consider the following example:
`withdraw(uint256)` get hashed to
`0x2e1a7d4d13322 . . . 0d5b69f16a49f`
The first four bytes are then `0x2e1a7d4d`. The bytes are the function selector, telling the contract which function to call. Now the arguments need to be encoded. A user wants to withdraw 10000000000000000 wei, in hex, this number gets represented as `0x2386f26fc10000`. Since it is a 256-bit integer, the serialized representation needs to be padded. Thus we arrive at the following data field for the function call `withdraw(10000000000000000)`:
`2e1a7d4d000 . . . 0002386f26fc10000`.

However, how would a user create a new contract? This problem requires a special kind of transaction. Transactions to create a contract are sent to a unique zero address. This address is not an EOA nor a contract address. This address can only be used as a destination to create addresses.

A contract-creating transaction must only contain the compiled byte code in the data field. Its only side-effect is the creation of a contract. The value field can contain ETH to provide a starting balance for the contract if needed.

Transactions are signed by signing the Keccak-256 hash of the RLP-serialized transaction data. Signing happens in the following way:

- Build the Transaction: The sender creates a transaction, specifying parameters such as the recipient's address, the amount of ETH to be transferred, any data to be included, the gas limit, gas price, and the nonce.
- RLP Encoding: The transaction data is then serialized using RLP encoding.
- Hashing: The serialized transaction data is then hashed using the Keccak-256 hash function, creating a 32-byte hash that uniquely represents the transaction.
- Signing: The sender then uses their private key to sign this hash. Signing is done using the ECDSA algorithm, which outputs a signature consisting of two parts, "r" and "s."
- Appending: To create a valid transaction, the "v", "r", and "s" values are added. The "v" value is added to aid recovery of the correct public key. This value is calculated as: $\text{CHAIN\_ID} * 2 + 35$ or $\text{CHAIN\_ID} * 2 + 36$.
- Recovering the sender's address: When a node receives a signed transaction, it can perform a signature verification process. Using the same ECDSA algorithm along with the provided signature (r and s values) and the transaction hash, the node can recover the sender's public key. This public key is then hashed, and the last 20 bytes of this hash are taken, resulting in the Ethereum address of

the sender. The signature is considered valid if this computed address matches the sender's address in the transaction.

- Transaction Propagation: Once the transaction is signed, it is broadcast to the Ethereum network, where it awaits inclusion in a future block by a miner.

The extra "v" value is necessary because, given two values, "r" and "s," nodes can generate two possible public keys. Thus a third value is needed to get certainty.

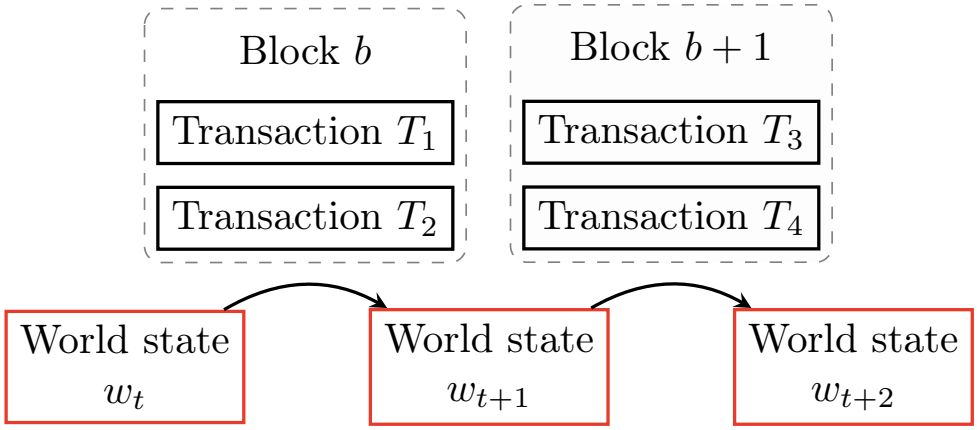


Figure 4.1: State changes to global state.

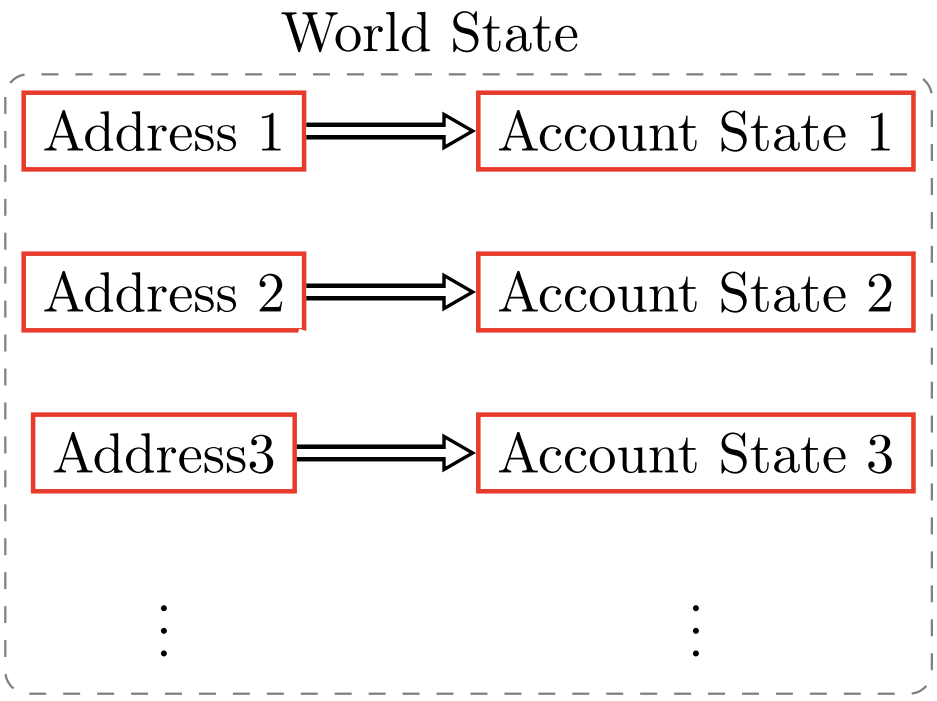


Figure 4.2: Contents of the world state.

4.5 Smart Contracts

As mentioned before, contract accounts by programs that the EVM executes. These programs are referred to as *smart contracts*. Where EOAs are controlled by transactions

signed by external private keys, contract accounts do not have private keys. They control themselves prescribed by their code. Both accounts have an address, nonce, and balance in common, as shown in figure 4.3. In the context of Ethereum, a smart contract is an immutable program that runs deterministically on the EVM as part of the Ethereum protocol. The storage of a smart contract is mutable since it records the program's state.

A contract can still be deleted by calling a special `SELFDESTRUCT` opcode. This opcode deletes the contract's code and storage and creates an empty account. Any transaction sent to the address does not result in any execution of any program. Calling the self-destruct opcode results in a refund of gas, thus incentivizing the release of resources and debloating the system. This

Smart contracts on Ethereum are typically written in a high-level called *Solidity*, which will be discussed in-depth in section 4.7. Other languages are available, but Solidity is the most common. In order to run, these programs have to be compiled to a low-level byte code which the EVM is able to interpret. Once compiled, contracts are deployed by sending a unique contract creation transaction to address `0x0`. After creation, each contract is identifiable by an address generated from the creation transaction, originating account, and nonce. This contract address can then be used as the recipient of a transaction to interact with the contract. Do note that the developer has to program a function to call the self-destruct opcode explicitly.

It is important to note that contracts are only ever run when the transaction calls them; contracts are thus reactive. A contract can, however, call another contract in its code. However, ultimately, the call originated from an EOA.

In some way, transactions are also atomic. Transactions only cause changes in the global state if they are successfully terminated. Successful termination means that a program is executed without causing any errors and without running out of gas. If execution fails in any way, all its effects are rolled back. The transaction never ran, but it is still recorded on the blockchain as an unsuccessful attempt at executing it. Ether spent on gas is also subtracted from the sender's balance.

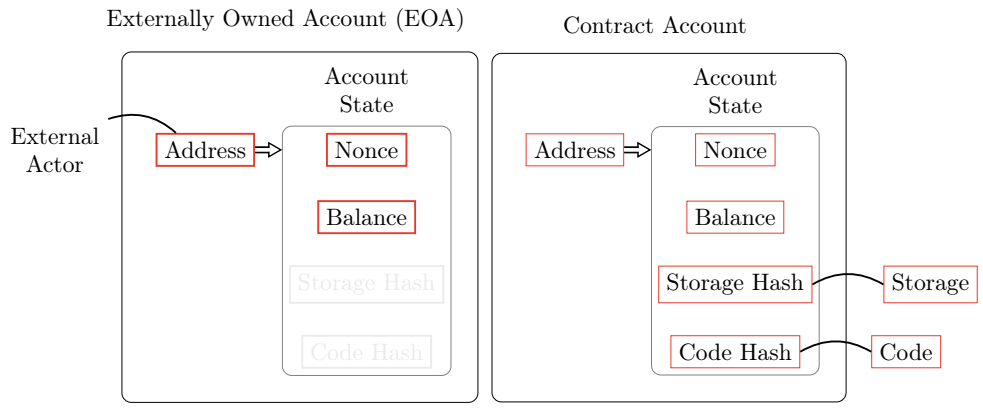


Figure 4.3: The two types of accounts in Ethereum.

4.6 Consensus

Ethereum, like Bitcoin, initially adopted a *proof-of-work*, PoW, consensus mechanism known as Ethash. Under PoW, network participants, also called miners, compete to solve a cryptographic puzzle, the difficulty of which is adjusted dynamically so that a new block is added approximately every 15 seconds. However, since the Ethereum 2.0

upgrade in 2022, it has used a *proof-of-stake*, PoS consensus algorithm, which will be discussed after the proof-of-work mechanism.

When a miner successfully solves the puzzle, they propose a new block to the network. The block contains a series of transactions, the previous block's hash, and the puzzle's solution. Other miners in the network verify the solution and, if correct, add the block to their local copy of the blockchain. This successful miner is rewarded with a predetermined amount of ETH (block reward) and any transaction fees from the transactions included in the block.

While PoW has effectively secured the Ethereum network, it is notoriously energy-intensive. PoW requires miners to perform computationally intensive tasks, most of which go to waste as only one miner's work is ultimately added to the blockchain.

Ethereum has been working on a significant upgrade known as Ethereum 2.0 to address these limitations. The centerpiece of this upgrade is a transition to a PoS consensus mechanism, dubbed "Eth2" or "Beacon Chain".

Under PoS, the block validation process is handled by validators, who are chosen to propose and attest to blocks based on the amount of ETH they hold and are willing to *stake* as collateral. Rather than competing in terms of computational power, the PoS model requires validators to have a financial stake in the network.

Validators are chosen pseudo-randomly to achieve the same randomness as PoW to create blocks, and other validators vote on their proposed blocks. The weight of each validator's vote depends on the size of its stake. If a validator tries to compromise the system, a portion or all of their staked ETH can be *slashed*, which means a portion will be removed. Slashing provides a strong disincentive for malicious behavior.

Proof-of-Stake offers several benefits over Proof-of-Work. It drastically reduces energy consumption as it does away with energy-intensive mining. It can offer better security by making a 51% attack more costly, as the attacker would need to own a large amount of ether, which would likely be rendered worthless after a successful attack. Furthermore, PoS opens the door to more sophisticated sharding mechanisms, which could significantly increase the network's transaction capacity.

Sharding aims to solve a scalability issue by splitting the Ethereum network into smaller pieces, called *shards*. Each shard would contain an independent state, meaning a unique set of account balances and smart contracts. Instead of every node storing and processing every transaction, nodes would be assigned to specific shards and would only process transactions and smart contracts within that shard.

Proof-of-work

The PoW mechanism Ethereum employs is known as Ethash. Ethash has been designed to resist the hardware optimization seen in Bitcoin's mining algorithm, thus maintaining a higher level of decentralization.

Proof-of-work in Ethereum, as with Bitcoin, involves miners competitively solving a complex mathematical puzzle, using the data from the latest block as input. The output, known as a nonce, must be less than a dynamically calculated target value to be considered valid. The competition to find this nonce is based on random guesses, making the likelihood of finding a valid nonce proportional to the miner's computational power relative to the rest of the network. Note that this nonce is at the block level and is not the same as the transaction nonce discussed previously.

Ethash is designed as a memory-hard algorithm, requiring a significant amount of memory to run, in addition to CPU and GPU resources. This memory-hardness is an intentional design feature to prevent the creation of Application-Specific Integrated

Circuits, or ASICs. ASICs could make the mining process more centralized, as with Bitcoin. Ethash achieves memory-hardness by requiring the storage and frequent access of a large dataset known as the Directed Acyclic Graph, abbreviated as DAG, during the mining process.

The DAG does not contain transaction data or block information. Instead, it comprises pseudorandom data, automatically generated every 30,000 blocks for about five days. The data in the DAG is generated from a simpler, smaller dataset called the seed. The seed is only 16MB and changes every epoch. The seed is used to generate the complete DAG, which was around 4 GB but started at 1 GB. The generation of the DAG involves repeated hashing and mixing of the seed data, which results in a large, pseudorandom data set.

In mining, a miner hashes the block's header and a nonce to get a result. This result is used as an index to select a DAG slice, which is hashed again. The second hash must be under a specific difficulty value for the block's validity.

Once a miner finds a nonce that satisfies the conditions of the mathematical puzzle, they package it along with the transactions they want to include and the hash of the last valid block. They then broadcast this proposed block to the Ethereum network.

Other miners in the network independently verify the validity of the proposed block, checking that the transactions are valid and that the nonce is correct. If they agree on the validity, they append the new block to their copy of the blockchain and begin mining the next block using the hash of the newly accepted block. The miner who found the valid nonce receives a block reward in ether and any transaction fees associated with the transactions they chose to include in the block.

Ethereum's *block time*, the average time it takes to mine a new block, is approximately 15 seconds, significantly faster than Bitcoin's ten minutes. The lower block time is achieved by adjusting the mining puzzle's difficulty to align with the network's total hashing power.

Proof-of-stake

Ethereum adopted a new consensus mechanism on the 15th of September, 2022, moving from proof-of-work to proof-of-stake. This adoption is known as the *Merge*. Initially, the *Beacon Chain*, a blockchain running a proof-of-stake protocol, ran in parallel with the mainnet, as seen in figure 4.4. The mainnet continued to be secured by proof-of-work. The Merge happened when these two parallel systems merged, and proof-of-work was permanently replaced by proof-of-stake.

Ethereum's proof-of-stake consensus protocol bolts together two different consensus protocols called *LMD GHOST* and the other *Casper FFG*. The combination has become known as Gasper.

Before diving into the details of Gasper, some preliminary terminology has to be explained.

Gasper introduces an idea called *finality*. Finality says that there are blocks in the blockchain history that will never be reverted. A *finalized block* is a block on which all honest nodes have agreed that will forever remain part of the history. Therefore, all of the block's ancestors too.

Bitcoin's consensus protocol has yet to have a concept of finality whatsoever. There will always be a possibility that a node will reveal a longer chain than the current one. When a node does this, all honest nodes must reorganize their chain and revert all processed transactions to their mempool. The amount of confirmations a block has on the Bitcoin blockchain is only an approximation of finality, not a guarantee.

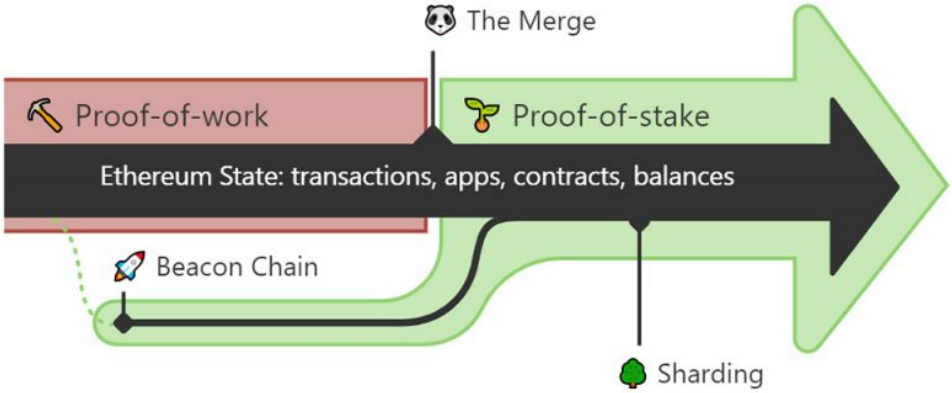


Figure 4.4: Visualization of merging the Beacon Chain [Eth23].

In Ethereum's consensus mechanism, finality is provided by Casper FFG. Casper FFG works by periodically having all validators agree on checkpoint blocks past which they will never revert. Then this block and all of its ancestors are final. It is possible to have conflicting checkpoints, but Casper FFG makes this cost enormous.

Two essential aspects of consensus mechanisms are safety and liveness. Safety means the system behaves like a centralized implementation that executes operations individually. The system should always be consistent. Liveness says that something good eventually happens. In the context of blockchain, this means that the chain can always add a new block.

The CAP theorem says that distributed system cannot provide consistency, availability, and partition tolerance. This theorem implies that designing a consensus mechanism that is both live and safe in all circumstances is impossible.

Ethereum thus prioritizes liveness. When a network gets split, both partitions will continue to produce blocks. Eventually, both partitions will finalize a different history, and the two chains will become independent forever.

Consensus is formed by *validators*. Each validator has an initial stake of 32 ETH. Validators also have a public and private key that identifies them in the protocol. Each validator is attached to a single node, and each node can possess multiple validators. These validators are not independent but share the same view.

A big difference from proof-of-work is that a complete list of participants in the consensus protocol is available. Accessing the available validators allows the protocol to achieve finality, as it is possible to know when a majority vote has been reached. A majority is here $\frac{2}{3}$ of the validators.

Time in the proof-of-stake consensus protocol is a significant shift from proof-of-work, where there was only a loose attempt at making block intervals regular. Time is divided into epochs and epochs into slots. Each slot is a 12-second window, and an epoch comprises 32 slots, amounting to 6.4 minutes. Figure 4.6 shows the division into timeslots and epochs.

During each slot, a random validator is picked to propose a block; this is the slot leader. This block contains updates to the beacon block and Ethereum transactions. A beacon block includes validator slashings, voluntary exits, and transfers, among other operations related to the functioning and security of the network. The proposer then shares its block with the entire network.

Do note that a slot can be empty. The selected validator may be offline or propose an invalid block. Ideally, this only happens sometimes, but the protocol is robust enough when empty slots happen.

Validators get to share their view on the chain's state every time an epoch ends. The sharing happens through attestations, a vote for the head of the chain, and a checkpoint. The LMD GHOST protocol uses the head of the chain and Casper FFG as the checkpoint. The attestations are also spread around the network and, like blocks, can be missing for the same reasons.

The attestations happen only once epoch to lighten the workload. Attesting is informing every validator of the view of every other validator. This spreading of information is a considerable amount of traffic and computing. Spreading the attestations over 32 slots keeps the traffic low as in each slot, only $\frac{1}{32}$ of the validators make attestations. These are called committees.

The randomness to select slot leaders and committees is provided by the *RANDAO*, a random number decentralized autonomous organization. The RANDAO works by having each member generate a random number independently. These random numbers are then mixed to ensure that each of these numbers have an equal influence on the outcome.

Each validator is required to provide a RANDAO reveal when they are proposing a block. Instead of revealing the actual number, the validators provide a hashed function. The hashing is necessary to prevent subsequent validators from influencing the random number created at the end.

At the end of the epoch, each validator reveals its random number. The random numbers are XOR'd to produce the final random number. This process ensures that validators can only predict or manipulate the final random number if they control most of the validators in a given epoch. The resulting number is then used to select slot leaders and committees. A visual representation of the RANDAO can be seen in figure 4.5.

Accuracy and block productions are incentivized by using rewards and penalties for validators.

In proof-of-work mining, a single block is costly. This scarcity incentivizes miners to follow the protocol's goals to ensure their block gets included.

Contrast this with proof-of-stake, where producing blocks and attestations is essentially free. Something is needed to prevent malicious actors from disrupting the protocol. *Slashing* provides a way to prevent these actions. Validators that hedge blocks or attestations are in danger of being slashed. Slashing is the process of ejecting validators and fining them by taking away a part of their stake. On the other hand, honest validators get rewarded for honest proposals and attestations. Hedging is doing contradictory things, like proposing two blocks or making two inconsistent attestations.

Now that all terminology is cleared up, Ethereum's consensus mechanism can be outlined.

Casper FFG has already been explained in the finalization section. Thus, only LMD GHOST and Gasper remain. LMD GHOST is a fork-choice rule.

Proof-of-work protocols use the longest-chain rule. The head block in the fork represents the fork that required the most cumulative work done.

LMD GHOST measures the weight of a fork not based on the work done but considers the latest attestations of the validators. Thus the chain with the most votes is the heaviest and is selected as the canonical chain. LMD GHOST incentivizes validators

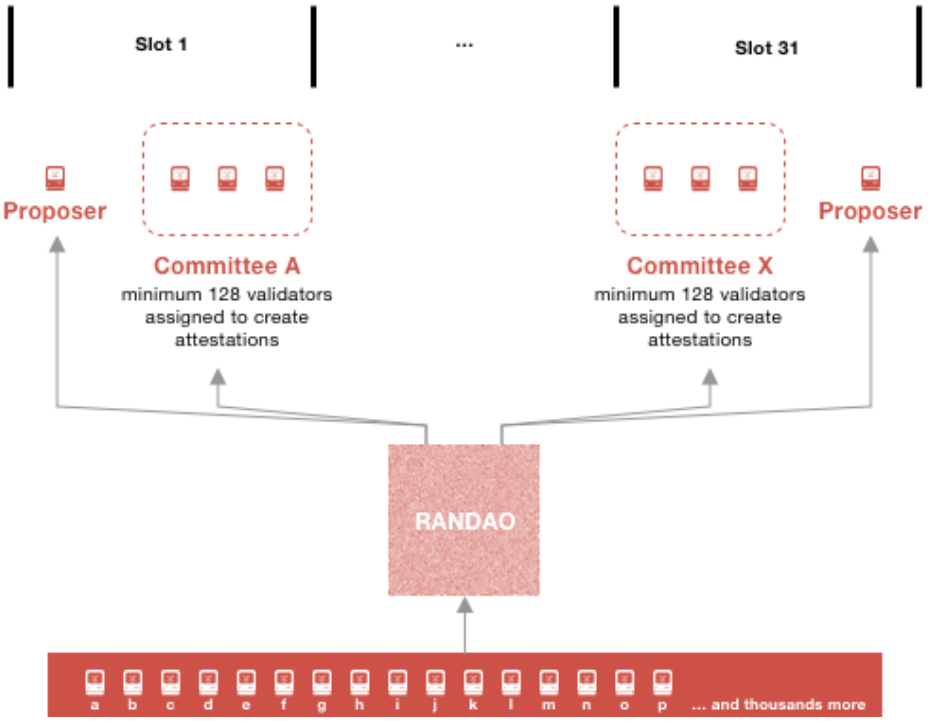


Figure 4.5: At every epoch, a pseudorandom process RANDAO selects proposers for each slot and shuffles validators to committees [Eth23].

to stay online and participate actively in the consensus process by focusing on the latest messages.

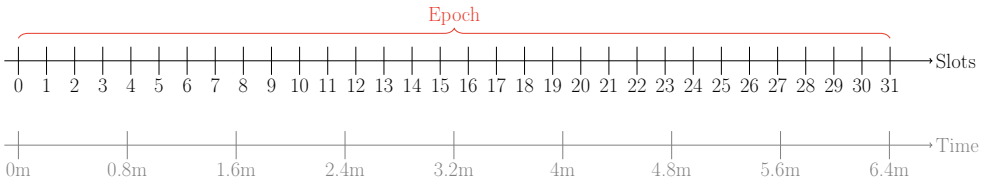


Figure 4.6: Slots and epochs in the proof-of-stake mechanism.

A high-level walkthrough of a single epoch is presented to clarify the consensus mechanism.

1. Slot assignment: before an epoch starts, validators are randomly assigned to slots in the upcoming epoch. For each slot, one validator is selected as the proposer, responsible for creating the block for that slot. Other validators assigned to the slot are attesters, who vote on the block's validity.
2. Block proposal: when each slot begins, the designated proposer creates a block containing data, such as attestations from the previous epoch and slashings. The block is then propagated to other validators.
3. Attestations: attester validators for the slot, then check the proposed block. If the validator concludes that the block is valid, they create an attestation for the block. This attestation includes information about the block itself, the current state of the chain, and the validator's signature. Attestations are then broadcast to the network.

4. Aggregation of attestations: attestations from individual validators are aggregated into "aggregated attestations" to improve efficiency.
5. Inclusion in Future Block: these aggregated attestations are then included in a future block within the same epoch or subsequent few epochs, propagated, and added to the chain.
6. Finality: blocks are then subject to a finality rule for the chain, Casper FFG. Once a block receives two-thirds of the votes in its favor within its epoch and the next one, it becomes finalized, meaning it can no longer be changed or reversed.
7. Rewards and Penalties: validators are rewarded for proposing blocks and for their attestations being included in the chain. Conversely, they are penalized for misbehavior or non-participation.

4.7 Solidity, A Contract-Oriented Language

Solidity is a statically-typed, contract-oriented programming language developed for implementing smart contracts on various blockchain platforms, notably Ethereum. It is statically typed and supports inheritance, libraries, and complex user-defined types, rendering it a reliable tool for designing contracts on the blockchain.

Ethereum's blockchain, with its support for programmable smart contracts, requires a specialized and secure programming language. Solidity provides developers with the necessary means to write decentralized applications in this context.

Solidity employs a contract-oriented programming paradigm, reflecting the transactional nature of the problems it is designed to solve. This design principle helps developers structure their code around contract entities, which encapsulate data and functions that manipulate this data. Each contract can be considered a self-contained code unit with clearly defined interfaces. This paradigm closely aligns with the blockchain's transaction-based model of computation, making Solidity a good choice for blockchain development.

It offers comprehensive features like type safety, control structures, function visibility specifiers, and error-handling mechanisms. In addition, it supports features like multiple inheritance, interfaces, and abstract contracts, to facilitate the more expressive and modular design of smart contracts.

The following sections delve deeper into the features of Solidity and dissect the syntactical and semantic elements that contribute to this language's unique and powerful functionality in the context of smart contract development on the Ethereum platform.

Solidity Programming Basics

First, the basic structure of a Solidity program will be discussed. Below is listing 4.1, a basic program is presented. This code is a simple faucet contract that provides ETH to any sender who requests it with a limit of 0.1 ETH per transaction. This code looks like a C++ or Java program. The following can be found in this small program:

- Lines one and two are the licensing and a pragma keyword. The pragma keyword is a compiler directive that specifies the compiler version to be used for this file.
- Line four is the contract name, here "Faucet". This contract can be seen as a class in C++.
- Line six contains a variable declaration. Here is a mapping from `address` to `uint256`. A mapping can be interpreted as a hash map, where the first mapping

type is the key and the second the value.

- Line eight defines a function with a parameter and return value. The public keyword means everyone can call the function.
- Line ten contains a requirement. This function is used to verify inputs and conditions before running the function.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  contract Faucet {
5      // Keep a list of addresses with the amount withdrawn
6      mapping(address => uint256) withdraws;
7
8      function withdraw(uint withdraw_amount) public return (true) {
9          // Limit withdrawal amount
10         require(withdraw_amount <= 100000000000000000);
11         // Add a withraw to list
12         withdraws[msg.sender] += withdraw_amount;
13         // Send the amount to the requested address
14         msg.sender.transfer(withdraw_amount);
15
16         return true;
17     }
18     // Accept any incoming amount
19     function () public payable {}
20 }
```

Listing 4.1: Faucet contract.

File Structure

Now that a minor contract has been examined, all possibilities of file contents will be discussed.

First is the pragma directive. The pragma below indicates that a below version 0.7.0 should not be used to compile the source code.

```
pragma solidity^0.7.0
```

It is also possible to constrain the compiler to a range of versions, as shown in listing 4.1:

```
pragma solidity >=0.7.0 <0.9.0;
```

Developers can also import from file names. Take the following file structure:

```
├── main.sol
└── foo.sol
```

Importing `foo.sol` into main can be done as follows:

```
pragma solidity^0.7.0
import "./foo.sol"
//rest of the file
```

It is also possible to import specific symbols using an optional alias from a file using the following syntax.

```
pragma solidity^0.7.0
import {contract1 as alias_name, contract2}
  from "./foo.sol"
// Rest of the file
```

Or import using a self-defined namespace.

```
pragma solidity^0.7.0
import * as symbol_namespace from "./foo.sol"
// Rest of the file
```

The comment style is the same as C/C++/Java/....

```
// Single line comment
/*
This is a Multi-line comment
*/
```

Contracts are similar to classes in object-oriented languages, as mentioned above. Contracts also have constructors, member functions, and member variables.

```
pragma solidity ^0.7.0;

contract example {
  uint256 _var;

  constructor(uint256 _initialVal) public {
    _var = _initialVal;
  }
  function setVar(uint256 input) {
    _var = input;
  }
  function getVar() returns(uint256) {
    return obj;
  }
}
```

Solidity also has libraries. Libraries are similar to contracts but cannot declare state variables, have payable functions, use fallback functions, or inherit.

Payable functions let a smart contract accept ether. They help developers manage incoming ETH and take action when ETH is received.

A contract can only have one fallback function. Fallback functions are executed on a call to the contract if none of the other functions match the given function signature or if no data was supplied and there is no payable function. Fallback functions can also receive ETH if they are marked as payable.

However, a library can modify the state variable in the contracts which call it. For example, call a library function from a contract and pass a state variable from the contract to the library function as a parameter. The variable can be modified in the library function, and the change will be reflected in the corresponding contract. This process is similar to passing a pointer in C or C++.

It is also possible to attach library functions to a specific datatype as follows:

```
using <libraryName> for <dataType>
```

Below is an example of a library and passing a state variable to it.

```
pragma solidity ^0.7.0;

library addLib {
  function addOne(uint256 input) {
    input += 1;
  }
}

contract example {
  using addLib for uint256;
  uint256 _var;

  constructor(uint256 _initialVal) public {
    _var = _initialVal;
  }

  function addVar() {
    _var.addOne();
  }
}
```

Like other object-oriented programming languages, Solidity also has prototypes in the form of interfaces.

```
pragma solidity ^0.7.0;

interface exampleInterface {
  function withdraw(uint withdraw_amount)
    public return (true);
}
```

Contract

Here the structure within a contract is briefly discussed. A contract can have the following components:

- State variables.
- Structure definitions.
- Modifier definitions.
- Event declarations.
- Enumeration definitions.
- Function definitions, including a payable and fallback function. A constructor can be considered a function too.

These components are shown in listing 4.2. Before diving deeper into each component, first, a short breakdown of each component.

- State Variables: Variables that persistently store data in contract storage.
- Structure: Structure allows the creation of custom data types.
- Enumerations: Enumerations help create user-defined types with a finite value set.
- Events: These give the EVM logging facilities an abstract interface.

- Modifiers: Modifiers can be used to change the behavior of functions in a declarative way.
- Constructor: This special function gets called once and only once when the contract is first created.
- Functions: Functions define the executable code associated with the contract.
- Fallback function: A contract can declare exactly one fallback function using the fallback keyword.
- Receive function: A contract can have exactly one receive function declared using the received keyword.

```
contract ContractName {
    // State Variables
    uint private stateVariable;

    // Struct Definition
    struct CustomStruct {
        uint id;
        string name;
    }

    // Enums
    enum CustomEnum {
        Option1,
        Option2
    }

    // Events
    event LogData(uint dataToLog);

    // Modifier
    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    // Constructor
    constructor() public {
        // 'msg.sender' is a special keyword that refers to the address of the entity
            ↪ (user or contract) interacting with the contract.
        owner = msg.sender;
    }

    // Functions
    function doSomething() public onlyOwner {
        // function logic goes here
    }

    // Fallback function
    fallback() external payable {
        // This function gets executed if no other function matches the called function
            ↪ and no receive ETH function exists.
        // It must be external and payable.
    }

    // Receive function
    receive() external payable {
        // This function is executed on a call to the contract with empty call data.
        // This is the function that is executed on plain ETH transfers.
    }
}
```

Listing 4.2: Structure of a Solidity contract.

These individual items are discussed in the following sections.

Variables

Just like traditional programming languages, in Solidity, there are basic data types. These include bool, int, uint, and struct. Floating points are missing and for good reason. In Solidity, developers deal with currencies, and due to the inherent rounding errors of floating point numbers, it is not suitable and even dangerous to use. Thus, all calculations use the smallest denomination of ether, wei, using 256-bit integers.

Regarding data types, Solidity differentiates between static-sized and dynamic-sized types. Static-sized types always consume the same amount of space in the EVM, no matter their value. Dynamic-sized types are types where the amount of space used can vary. When one talks about static and dynamic in Solidity, they usually refer to the size of the data types and how they are stored and handled in the EVM, not the value of the variables.

The value of variables in Solidity can change unless they are declared as *constant* or *immutable*. A constant variable can be set at compile time, while an immutable variable can be set once during contract creation and then becomes read-only. Apart from these, variable values can be changed in Solidity.

Boolean types, `bool`, can be true or false. Unlike C-like languages, booleans cannot be implicitly cast to an integer. Explicit casting is allowed, however.

Integer and unsigned integer types have different sizes. They go from (u)int8 . . . (u)int256 in steps of eight bits. `int` and `uint` are aliases of `int256` and `uint256` respectively. The language has built-in overflow and underflow protection since Solidity version `0.8.0`. Before this, the developer had to write his own "SafeMath" library or an existing one.

The `address` type is a 20-byte long data type and is used to store Ethereum addresses. Addresses do not allow any arithmetic operations but can be converted to `uint160` or `bytes20`. There is also an `address payable` type to send ETH.

The next static-sized type is fixed byte arrays, declared as `bytesN` where `N` is one . . . 32. Hexadecimal or decimal literals can be used to set the byte arrays:

```
bytes1 x = 0x75;
bytes1 y = 10;
bytes1 z = -100;
bytes2 k = 256;
```

For byte types there exist bit operations: `&`, `|`, `^`, «, and ». Comparisons and index access can also be used.

As far as literals go, the following literals exist:

- Integer literals: simply natural numbers.
- String literals: surrounded by single or double quotes.
- Address liters: prefixed by `0x`, for example `0xfa35b7d915458ef540ade6078dfe2f44e8fa733c`.
- Hexadecimal literals: strings using 0x as prefix and `hex` before quotations. For example, `hex"0x9A5C2F"`. Digital item literals: these are decimal fractions, and scientific notation can also be used. For example, `7.5` and `2e10`.

Enumerations also exist. These allow developers to create a user-defined type with a limited range of values which are also explicitly convertible to integers.

```
contract test {
  enum CustomEnum {
      Option1,
      Option2
  }
  CustomEnum choice = CustomEnum.Option1;
  uint choiceNum = uint(choice);
}
```

For more complex types, Solidity uses reference types. Reference types do not hold the actual data stored in a variable but a reference or a pointer to the location of this data. When a reference type is assigned to another, it does not create a new copy of the data but simply points to the original data.

Reference types also have an additional *data location* annotation. This annotation gives information about where the variable is stored. There are three possible locations: `memory`, `storage`, and `calldata`.

Storage is where state variables reside. Every contract has its storage, which is persistent between function calls and transactions. Variables in storage are relatively expensive to use in terms of gas cost, given the necessity of preserving data on the blockchain.

On the other hand, memory is a data area that is temporarily used during function execution. It is erased between external function calls and is cheaper to use regarding gas cost. Memory arrays in Solidity are used for temporary storage within a single function call. Any changes made in a memory array do not persist once the function has finished executing.

Call data is an immutable, volatile area where function arguments are stored and behaves mostly like memory.

It is important to note that reference types behave differently in memory, temporary space during execution, and storage, permanent space on the blockchain. In storage, assignment between reference types creates a copy, but in memory, it only assigns a reference. Assignments between storage and memory or from call data always create a copy. Understanding these details is crucial for writing efficient Solidity code, as storage operations can be significantly more expensive than memory operations in terms of gas costs.

Arrays can have a compile-time fixed size or a dynamic size. A fixed-size array is declared like this: `A[7]`, and a dynamic-sized array like this: `A[]`; Accessing an array past its end causes a failing assertion. The methods `.push()` and `.push(value)` can be used to append a new element at the end of the array.

Variables of type `bytes` and `string` are special arrays; the `bytes` type is similar to `bytes1[]`, but it is packed tightly in call data and memory. Strings are similar to bytes but do not allow length or index access.

We can use the keyword "new" to create a memory type array. The `new` keyword is used to create a new memory array. The significant difference between memory and storage arrays is that memory arrays cannot be resized, but storage array can. This difference is visible in the following example:

```
contract example {
  function f() {
    // Create a memory array
    uint[] memory a = new uint[](7);
    // We can not update the length property of
    // memory array
    // Error
    // a.length = 100;
  }
  //storage Array
  uint[] b;
  function g(){
    b = new uint[](7);
    // We can adjust array length by setting
    // length property
    b.length = 10;
    b[9] = 100;
  }
}
```

Like in other languages, structures allow users to define their composite datatypes.

Mappings can be thought of as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte representation are all zeros.

Functions in Solidity are first-class citizens who can be assigned to variables, pass a function as a function parameter, and return a function from a function. A complete function definition is the following:

```
function <functionName>(<parameters>) <visibility>
<functionModifiers> <returns(<returnTypes>)> {
  <functionBody>
}
```

- <functionName>: this is the name of the function.
- <parameters>: this is a comma-separated list of parameters the function accepts. Each parameter is represented as <type> <parameterName>. For instance, `uint _value`.
- <visibility>: this determines who can call the function. The options are `public`, `external`, `internal`, and `private`. Public and external functions can be called from outside the contract, but external functions can only be called from other contracts, not from within the contract itself. Internal and private functions can only be called from within the contract.
- <functionModifiers>: This is a space-separated list of function modifiers, conditions checked before the function is executed. Some examples are view, pure, payable, and constant.
- <returns(<returnTypes>)>: This is the type of data the function will return. If a function does not return anything, this part can be omitted.
- <functionBody>: This is the actual code of the function.

Take the following functions, for example. Consider the following function type:

```
function (uint16, uint32) view returns (uint64)
```

Variables of this type can be assigned functions:

```
function add(uint16 x, uint32 y) internal
pure returns (uint64) {
  return uint64(x) + uint64(y);
}
function mul(uint16 x, uint32 y) internal
pure returns (uint64) {
  return uint64(x) * uint64(y);
}
function runFunc(bool flag) external
pure returns (uint64) {
  function (uint16, uint32) view returns (uint64) func
    = flag ? add : mul;
  return func(x, y);
}
```

Function parameters of the type can be used for passing functions to functions:

```
function receiveFunc(function (uint16, uint32) view
returns (uint64) func) external pure returns (bool) {
  if (func == add)
  return true;
  else
  return false;
}
```

Function parameters of function type can be used for returning functions from functions:

```
function returnFunc(bool flag)
external pure returns (function (uint16, uint32)
  view returns (uint64)) {
  if (flag)
  return add;
  else
  return mul;
}
```

Please note that Solidity uses function modifiers like view, pure, payable to denote function behavior:

- View: this function promises not to modify the state.
- Pure: this function promises not to read from or modify the state.
- Payable: this function sends ETH to the contract.

Solidity also contains some predefined variables and functions which exist in the global namespace. When the EVM executes a contract, it has access to a set of global objects. These objects include the `block`, `msg`, and `tx` objects. Below is a list of these functions and variables.

- `msg.sender`: contains the address that initiated the contract call. It can be an EOA or contract address.
- `msg.value`: the value, in wei, sent with this call.
- `msg.data`: the data of this call.
- `msg.sig`: the first four bytes of the data, which is the function selector.

- `tx.gasprice`: the gas price defined in the transaction.
- `tx.origin`: the address of the originating EOA for this transaction.
- `block.coinbase`: the address of the block's validator.
- `block.prevrandao`: the random number provided by the Beacon Chain.
- `block.gaslimit`: the gas limit of this block
- `block.number`: the current block number, also known as the blockchain height.
- `block.timestamp`: the timestamp of the current block.
- `gasleft()`: remaining gas in the current call.
- `blockhash(uint blockNumber)`: hash of the given block, the provided number must be one of the 256 most recent blocks.
- `addmod`, `mulmod`: modulo addition and multiplication.
- `keccak256`, `sha256`, `sha3`, `ripemd160`: hash functions for various standards.
- `selfdestruct(recipient_address)`: deletes the current contract and sends the remaining ETH to the recipient address.

Statements

Solidity does not support switch and goto statements. However, Solidity does support the other standard control flow statements. These include *if-else*, *while*, *do-while*, *for*, *break*, and *continue*.

Modifiers

In Solidity, modifiers are code snippets that can be run before and after a function call. They are used to modify the behavior of a function and can be used to verify certain conditions before executing a function, leading to cleaner and safer code. Modifiers can be used to automate checks that apply to multiple functions, reducing the amount of duplicated code and, thereby, potential errors.

A modifier declaration looks just like a function declaration, but with the keyword `modifier` instead of `function`, and they cannot return anything:

```
modifier <modifierName>(<parameters>) {
  <modifierBody>
}
```

A modifier's body may contain a special statement `_`. This underscore is where the execution of the original function is inserted. The function call will not be executed if `_` is omitted.

Here is an example of a common modifier:

```
modifier onlyOwner() {
  require(msg.sender == owner,
    "Only the contract owner can call this function.");
  _;
}
```

In this example, the `onlyOwner` modifier will only allow the function to proceed if the contract owner calls it. If not, it reverts the transaction. A function can then use the modifier with its name after the function signature:

```
function transferOwnership(address newOwner) public
onlyOwner {
  owner = newOwner;
}
```

In this example, the `onlyOwner` modifier restricts access to the `transferOwnership` function. This function changes the owner state variable of the contract, which should not be accessible by just any user.

Multiple modifiers can be used on a function, and they will be applied in the order they are written:

```
function doSomething() public onlyOwner onlyWhenOpen {
  // ...
}
```

In this example, the function `doSomething` will first check the `onlyOwner` modifier, then the `onlyWhenOpen` modifier. If both pass, the function will be executed.

When multiple modifiers on a function, each modifier containing the underscore, _ symbol, is essentially wrapped around the function body. Each _ is replaced by the next modifier or the function body itself.

Consider the following:

```
modifier mod1 {
  ...
  _;
  ...
}

modifier mod2 {
  ...
  _;
  ...
}

function foo() mod1 mod2 {
  ...
}
```

This program would execute in the following order:

- The code before _ in mod1.
- The code before _ in mod2.
- The body of foo.
- The code after _ in mod2.
- The code after _ in mod1.

Events

Events in Solidity are an essential feature that facilitates communication between smart contracts and their external environment. They are a convenient tool that allows for the logging and listening of state changes in a contract to the Ethereum blockchain. When a transaction completes, it generates a transaction receipt. This receipt contains log entries that provide information about everything that happened

during the execution of the transaction. *Events* are the Solidity object to construct these logs.

Services can watch for specific events and report them to the user or other programs to then reflect events in an underlying contract. Event objects take serialized arguments and are then recorded in the transaction logs.

Events are declared with the `event` keyword followed by the event name and the argument types. Here is an example of an event declaration:

```
event ValueChanged(address indexed author, uint oldValue,
  uint newValue);
```

In this example, `ValueChanged` is an event that can be emitted with an address and two unsigned integers. The `indexed` keyword indicates that the author can be used as a filter parameter in the frontend application.

Events are emitted using the `emit` keyword followed by the event name and the data to log. Here is an example:

```
function setValue(uint newValue) public {
  uint oldValue = value;
  value = newValue;
  emit ValueChanged(msg.sender, oldValue, value);
}
```

In the `setValue` function, whenever the value state variable is changed, the `ValueChanged` event is emitted with the sender's address and the old and new values.

Events are added to the transaction receipt log, a data structure linked to each transaction. They can be later consumed from an off-chain service like a client application, which can use the data to react to the state changes. Applications can react to blockchain transactions in real time with events.

Accessing stored data on the blockchain can be costly in terms of gas. Events provide a way to return data from a contract function without needing to store it. This data is logged on the blockchain and can be read for a lesser cost. Every byte in a log costs eight gas, while each contract storage variable costs 20000 gas every 32 bytes. One downside is that logs are not accessible from other contracts.

Error Handling

In any programming environment, handling errors and exceptions is crucial for ensuring the reliable and secure execution of the code. In Solidity, this is especially important because once a contract is deployed on the Ethereum network, it cannot be altered, and errors can have a significant impact due to blockchain transactions' immutable and public nature.

In traditional programming, error handling often uses try-catch blocks or returns error codes from functions. However, these approaches are only sometimes used in Solidity due to the unique constraints of smart contracts, such as the need for atomic transactions and the high cost of computation and storage on the Ethereum network.

Instead, Solidity uses state-reverting exceptions to handle errors. When such an exception is thrown, all changes to the state made by the current function and all its sub-calls are reverted, and the remaining gas is returned to the caller. This revert ensures the atomicity of transactions, which means they either complete successfully or have no impact on the state.

The `revert()` function can throw an exception and revert the current state changes. It can also provide a reason string.

```
revert("This is the revert reason.");
```

The `require()` function is used to validate inputs and conditions. If the condition inside `require()` evaluates to false, an exception is thrown, and the code execution stops. It can also provide a reason string like `revert()`.

```
require(x > 0, "x must be greater than 0");
```

The assert() function is used for internal errors and checks on invariants. Failing assertions mean that there is a bug. Unlike `require()`, `assert()` does not facilitate providing a reason string.

```
assert(x != 0);
```

Custom errors can be defined and used in place of reason strings. They are defined using the `error` keyword and can have named parameters.

```
error InsufficientBalance(uint256 available,
  uint256 required);

function withdraw(uint256 amount) public {
  if (amount > balance[msg.sender])
  revert InsufficientBalance(balance[msg.sender], amount);
  balance[msg.sender] -= amount;
}
```

Inheritance

Just like other object-oriented languages, Solidity contracts support inheritance, which includes multiple inheritance. Contracts can inherit other contracts, gaining access to their state variables, events, modifiers, and functions, except those explicitly marked as `private`.

The basic syntax for contract inheritance uses the keyword. Here is an example:

```
contract Base {
  uint x;
  constructor(uint _x) {
      x = _x;
  }
  // Base contract code...
}

contract Derived is Base {
  constructor(uint _x) Base(_x) {
    // ...
  }
  // Derived contract code...
}
```

Derived contracts can override function implementations of their base contracts by declaring a function with the same name and parameters. The keyword `override` must be used in the derived contract's function:

```
contract Base {
  function foo() public virtual returns (string memory) {
    return "Base";
  }
}

contract Derived is Base {
  function foo() public override returns (string memory) {
    return "Derived";
  }
}
```

In this example, `Derived` overrides the `foo` function from `Base`. To call the overridden function, the `super` keyword can be used, which refers to the immediate parent contract:

```
function foo() public override returns (string memory) {
  return super.foo();
}
```

Solidity supports multiple inheritance. Multiple inheritance can introduce ambiguity, called the diamond problem. The diamond problem says that if two or more base contracts define the same function, which should be called in the child contract? Solidity deals with this ambiguity by using reverse C3 Linearization, an algorithm used to obtain the order in which methods should be inherited, which sets a priority between base contracts. That way, base contracts have different priorities, so the order of inheritance is relevant.

```
contract Base1 {
  // Base1 contract code...
}

contract Base2 {
  // Base2 contract code...
}

contract Derived is Base1, Base2 {
  // Derived contract code...
}
```

Contracts Calling Contracts

In Solidity, contracts can interact with one another through function calls. These calls can be within the same contract or between different contracts. This operation is beneficial but potentially dangerous. The risks arise because the caller may not know much about the contract being called. There is nothing to stop arbitrarily complex and malign contracts from falling into and being called by other code.

The safest way is to create an instance of the contract. This way, the user knows which interfaces and behavior are being used. To do this, the user can instantiate it using the `new` keyword. The `new` keyword will create the contract on the blockchain and return a reference to the object.

The following example creates an instance of the `TargetContract` contract:

```
import "TargetContract.sol";

contract Caller {
  TargetContract _target;

  constructor() {
    _target = new TargetContract();
  }
}
```

Then functions in the target contract can be called using this instance:

```
uint result = _target.targetFunction(123);
```

Note that while `Caller` is the owner of the `TargetContract` contract, the contract itself owns the new `Caller` contract, so only the `TargetContract` contract can destroy it.

The second way to call another contract is by using the address of an existing instance and casting it. In this way, the user applies a known interface to an existing interface. This method makes it crucial for the user to know that the instance being addressed is what he assumes it is. Take the following example:

```
import "TargetContract.sol";

contract Caller {
  TargetContract _target;

  constructor(address _t) {
    _target = TargetContract(_t);
    uint result = _target.targetFunction(123);
  }
}
```

In this example, the address is provided to the constructor and cast to the `TargetContract` object. This method is much riskier because the user only assumes the address is the correct object. The user needs to determine whether the function accepts the same arguments or executes the correct code. Thus this method is much more dangerous than creating a contract with the caller.

The last and most dangerous method for calling other contracts are low-level functions. These functions correspond to EVM opcodes of the same name and allow users to construct a contract-to-contract call manually. As such, these are also the most flexible methods. Here is an example using the `call` method:

```
contract Caller {
  constructor(address _t) {
    _t.call("targetFunction", 123);
  }
}
```

This program contains a blind call into a function. It exposes the contract to several security risks, the most important of which will be discussed in the *security* section of this chapter. However, the `call` function will return `false` if there is a problem. Thus the return value can be checked:

```
contract Caller {
  constructor(address _t) {
    if(!_t.call("targetFunction", 123)) {
      revert("Call to target failed!");
    }
  }
}
```

Another variant of `call` is `delegatecall`. A `delegatecall` differs from a `call` because the context does not change. For example, whereas a `call` changes the value of `msg.sender` to be the calling contract, a `delegatecall` keeps the same `msg.sender` as in the calling contract. In principle, `delegatecall` runs the code of another contract inside the context of the execution of the current contract. The delegate call should be used with great caution. It can have unexpected effects, especially if the called contract was not designed as a library.

Interacting with other contracts can open up several security risks. The called contract could execute malicious code, consume all the calling contract's gas, or cause the calling contract to revert. Therefore, it is crucial to sanitize inputs, limit gas, handle errors, and not rely on the order of transactions.

Smart Contract Design and Development

Now that the basics of Solidity have been discussed, the design of a smart contract can be explained. Designing a smart contract requires careful consideration of functionality, security, and efficiency. An everyday use case in Ethereum is the creation of ERC-20 tokens, a standard interface for fungible tokens.

In Ethereum, tokens can represent almost anything. Some examples include:

- Lottery tickets.
- Financial assets.
- A currency like the Euro.
- An ounce of gold.

ERC-20 defines the standard for creating these tokens, which are interoperable with other services.

This section will walk through designing a simple ERC-20 token contract.

The first step in designing a smart contract is to define the contract specification. For an ERC-20 token, the specification includes:

- A name for the token.
- A symbol for the token.
- Decimals defining the smallest unit of the token.
- The total supply of tokens.
- A mapping to store each Ethereum address's token balance.
- A mapping to allow addresses to approve others to spend tokens on their behalf.

The ERC-20 standard defines a set of functions and events which much be implemented. If a smart contract implements this set, it can be called an ERC-20 token contract. Once deployed, it will be responsible for keeping track of the created tokens on Ethereum. Since this set of functions and events have to be implemented, it is possible to define an interface for the ERC-20 standard:

```solidity
interface ERC20Interface {
    // Functions
    function name() public view returns (string)
    function symbol() public view returns (string)
    function decimals() public view returns (uint8)
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount) external
        ↪ returns (bool);
    // Events
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

Listing 4.3: ERC-20 interface.

Now the contract can be implemented. First, the state variables need to be set up. The tokens need a name, symbol, decimals, and total supply. The name, symbol, and supply are self-explanatory. The decimals variable specifies the smallest divisible unit of a token. Here 18 will be used, meaning each token can be divided into 10^{18} parts. This decimal is necessary because Solidity only supports integer numbers because of the previously mentioned accuracy issues. Here are the state variables defined in the contract:

```solidity
string  private constant name     = "TestToken";
string  private constant symbol   = "TTK";
uint8   private constant decimals = 18;
uint256 private totalSupply;
```

Now two mappings have to be created. One to record the balances of each address and one to record how many tokens a spender can spend on behalf of the owner. These mappings look like this:

```solidity
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256))
  private _allowances;
```

The first function is the constructor of the contract. It initializes the total supply and assigns all tokens to the contract deployer. Here the initial supply is multiplied by 10^{18} to allow the deployer to pass a human-readable argument to the constructor.

```solidity
constructor(uint256 initialSupply) {
  totalSupply = initialSupply * (10 ** uint256(decimals));
  // Mint all tokens to the contract deployer
  _balances[msg.sender] = totalSupply;
}
```

Next are the getters for an account's name, symbol, decimals, and balance. These are self-evident.

```
function name() public view returns (string) {
  return name;
}

function symbol() public view returns (string) {
  return symbol;
}

function decimals() public view returns (uint8) {
  return decimals;
}

function balanceOf(address account)
  public view returns (uint256) {
  return _balances[account];
}
```

The following functions transfer tokens from one account to another. The public `transfer` function allows the sender to transfer a specific amount of tokens to a recipient. When the `transfer` function is called, it, in turn, calls the `_transfer` function, passing in the sender's address in addition to the other parameters, which can be accessed thanks to `msg.sender`. The internal `_transfer` function is used by other functions to transfer tokens from one address to another. The `require` function calls are used to ensure that neither the sender's nor recipient's address is the zero address, `0x0`. If either address is the zero address, the transaction is reverted. An explicit underflow or overflow check is not necessary since Solidity `0.8.0`. Finally, the emit `Transfer(sender, recipient, amount)` line emits a `Transfer` event, which is useful for external watchers.

```
function transfer(address recipient, uint256 amount)
  public returns (bool) {
  _transfer(msg.sender, recipient, amount);
  return true;
}

function _transfer(address sender,
                   address recipient,
                   uint256 amount) internal {
  require(sender != address(0),
          "ERC20: transfer from the zero address");
  require(recipient != address(0),
          "ERC20: transfer to the zero address");

  _balances[sender] -= amount;
  _balances[recipient] += amount;
  emit Transfer(sender, recipient, amount);
}
```

The following functions are all involved in the process of allowing a third-party, such as a smart contract, to spend tokens on behalf of the owner. Here is a breakdown of all the functions.

The `allowance` function is a function that returns the current allowance the owner has given to the spender. The allowance is the number of tokens the spender is allowed to transfer from the owner account.

The approve function is called by the owner, accessed via `msg.sender` of the to-

kens, which wants to allow the spender to transfer a certain amount of tokens from the owner's account. When this function is called, it triggers the internal function `_approve`.

The `transferFrom` function is used by a spender to transfer several tokens from the sender's account to the recipient's account. The spender must have an allowance from the sender greater than or equal to the amount. After the transfer, the allowance is decreased by the amount transferred. Again, no underflow handling is necessary because of the previously mentioned built-in checking in the integer datatypes.

The `_approve` function is an internal function that sets the number of tokens the spender is allowed to transfer from the owner account. This function is called by `approve` and `transferFrom` to set or update the allowance. The function also emits an Approval event to log the approval on the blockchain.

```
function allowance(address owner, address spender)
  public view returns (uint256) {
  return _allowances[owner][spender];
}

function approve(address spender, uint256 amount)
  public returns (bool) {
  _approve(msg.sender, spender, amount);
  return true;
}

function transferFrom(address sender,
                      address recipient,
                      uint256 amount)
  public returns (bool) {
  _transfer(sender, recipient, amount);
  _approve(sender, msg.sender,
           _allowances[sender][msg.sender] - amount);
  return true;
}

function _approve(address owner,
                  address spender,
                  uint256 amount) internal {
  require(owner != address(0),
          "ERC20: approve from the zero address");
  require(spender != address(0),
          "ERC20: approve to the zero address");

  _allowances[owner][spender] = amount;
  emit Approval(owner, spender, amount);
}
```

Below, in listing 4.4, is the final contract for an ERC-20 token. Here all previously discussed functions are present to satisfy the standard.

```
1  pragma solidity ^0.8.0;
2
3  import "./ERC20Interface.sol";
4
5  contract TestToken is ERC20Interface {
6      string private constant name = "TestToken";
```

```
    string private constant symbol = "TTK";
    uint8 private constant decimals = 18;
    uint256 private totalSupply;

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    constructor(uint256 initialSupply) {
        totalSupply = initialSupply * (10 ** uint256(decimals));
        // Mint all tokens to the contract deployer
        _balances[msg.sender] = totalSupply;
    }

    function name() public view returns (string) {
        return name;
    }

    function symbol() public view returns (string) {
        return symbol;
    }

    function decimals() public view returns (uint8) {
        return decimals;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        _approve(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount) public
        ↪ returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, msg.sender, _allowances[sender][msg.sender] - amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }

    function _approve(address owner, address spender, uint256 amount) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
```

```
}
```

Listing 4.4: ERC-20 token implementation [Zhe+20].

This token only has the essential functions required to satisfy the ERC-20 standard. Extra features can be added. For example, minting and burning tokens and ensuring only the contract owner can mint and burn tokens.

Minting is the process of creating new tokens. The rules of minting are defined in the smart contract of the token. When the mint function is called, new tokens are created, and their ownership is assigned to a specified address. Minting increases the total supply of the token.

Burning is the opposite of minting. Burning destroys tokens, effectively reducing the total number of tokens in existence. Burning is typically done by sending tokens to a designated "burn" address from which they can't be recovered or spent. The effect of burning tokens is that it decreases the total supply of the token.

```
pragma solidity ^0.8.0;

import "./ERC20Interface.sol";

contract TestToken is ERC20Interface {
    string private constant name = "TestToken";
    string private constant symbol = "TTK";
    uint8 private constant decimals = 18;
    uint256 private totalSupply;
    address private _owner;

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    constructor(uint256 initialSupply) {
        _owner = msg.sender;
        _mint(_owner, initialSupply * (10 ** uint256(decimals)));
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "Caller is not the owner");
        _;
    }

    //rest of the functions described above

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) public onlyOwner {
        _burn(from, amount);
    }

    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");

        totalSupply = totalSupply + amount;
        _balances[account] = _balances[account] + amount;
        emit Transfer(address(0), account, amount);
    }

    function _burn(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero address");

        _balances[account] = _balances[account] - amount;
        _totalSupply = _totalSupply - amount;
```

```
48          emit Transfer(account, address(0), amount);
49      }
50  }
```

The contracts described in this chapter were deployed and tested using *Foundry*[Tea23] within a local, private blockchain.

Private blockchains are predominantly used for development and testing. These networks allow developers to deploy and interact with smart contracts instantly and at no cost.

There are also different public networks available for running and testing contracts.

The Ethereum *mainnet* is the primary public blockchain. The mainnet is where genuine ETH transactions occur and legitimate smart contracts are deployed. All actions performed on the mainnet are irreversible and permanently recorded on the blockchain, with associated costs paid in real ether. On this network, transactions with real ETH and value occur.

There are also Ethereum *testnets*. Testnets function as alternative, parallel blockchain networks specifically crafted for testing purposes. These networks emulate the mainnet environment but use distinct, valueless digital assets. Developers primarily use these public blockchain networks to test protocol upgrades and smart contracts before deploying them to the mainnet. This difference is analogous to a production versus staging server.

There are two types of testnets: permission proof-of-authority consensus testnets. In these testnets, a limited number of elected nodes validate transactions and create new blocks. The other type uses proof-of-stake, like the mainnet, where everyone can run a validation node.

At the time of writing, there are two maintained testnets:

- Sepolia: This is the testnet for application development. Sepolia uses a permissioned validator system. The network is small, so it quickly synchronizes and requires less storage. This network is helpful for users who want to start a node and interact with the network quickly.
- Goerli and Holešky: Goerli is being deprecated in 2023 and being replaced with Holešky. These networks are staking infrastructure and protocol developer testnets. Which means they are helpful for testing, validating, and staking. Any user can run a validating node to test staking. The state of the network is larger than the Sepolia network, which means synchronizing and setting up nodes is slower.

To get ETH on a test network, developers can use a faucet. A *faucet* is an application that gives away cryptocurrencies. These can also exist on main networks, but these only give away cryptocurrencies in exchange for completing simple tasks. In the case of test networks, these faucets are intended to supply cryptocurrencies to test applications. As the name faucet implies, these applications give away a few "drops" of cryptocurrencies periodically.

Gas Optimization

Gas usage is essential to consider when programming smart contracts. Gas is used to constrain the amount of computation a transaction can consume. If the gas limit is overrun, the following happens:

- An out-of-gas exception is thrown.

- The last state of the contract is restored.
- All ETH used to pay for gas is consumed and not refunded.

Gas is paid for by the user who creates the transaction and calls the function. Thus, users are not inclined to call functions with a high gas cost. This inclination means that the programmer has to optimize the gas usage of his contracts. This restriction is akin to embedded programming, but other aspects of computation are constrained instead of limited hardware resources. This section will recommend some practices when constructing smart contracts.

A loop through a dynamically-sized array increases the risk of running out of gas. Sometimes even before finding the desired element, thus wasting a lot of ether.

Calling other contracts, especially when the gas usage of their functions is unknown, also introduces the risk of running out of gas. Avoid libraries and contracts that are not well-tested and enjoy wide adoption.

Estimate the gas cost of each function being used in contracts. This estimation can be achieved using off-chain Ethereum libraries such as *web3py* for Python or *Geth* for Golang. Do note that it will always be an estimate because of the Turing completeness of the EVM. Evaluating gas costs regularly during development is recommended to avoid any surprises when deploying contracts.

Chapter 5

Cardano

Cardano, developed by IOHK, Input Output Hong Kong, represents an attempt at improving a blockchain system's fundamental architecture and capabilities. The project was initiated in 2015 with the intention of creating a more balanced and sustainable ecosystem for cryptocurrencies, built upon a scientific philosophy and rigorous academic research. This scientific and research-oriented approach to Cardano's development represents a divergence from the inception of many blockchain projects, which often lack such systematic development strategies.

Cardano was designed to rectify the blockchain ecosystem's three key challenges: scalability, interoperability, and sustainability. Scalability relates to the ability of a system to handle an increasing amount of work by adding resources to the system. In the context of Cardano, scalability pertains to network bandwidth, data storage, and transaction processing speed.

Interoperability, the second key challenge, refers to how blockchain systems communicate and interact. Given the multitude of blockchain platforms, interoperability is vital for information exchange and communication.

The final challenge, sustainability, pertains to how the blockchain system can be maintained and improved over time. For Cardano, this has led to the development of a treasury system, wherein a fraction of all transaction fees are directed towards a treasury to ensure future development and maintenance.

Cardano's blockchain is characterized by a dual-layered structure, which consists of the *Cardano Settlement Layer*, CSL, and the *Cardano Computation Layer*, (CCL). The CSL primarily deals with the operations of the ADA cryptocurrency, Cardano's native token, including tracking transactions. The CCL, the other hand, is responsible for executing smart contracts and computing transactions. This segregation of functions allows for a high degree of flexibility and adaptability.

Another noteworthy aspect of Cardano is its consensus mechanism, known as the *Ouroboros* proof-of-stake. This approach distinguishes itself from the more traditional proof-of-work system employed by Bitcoin and other early cryptocurrencies. The PoS system provides an energy-efficient mechanism through which holders of the ADA token can generate new blocks and validate transactions.

The Cardano project was started by Charles Hoskinson and Jeremy Wood, who were previously involved in the Ethereum project. Their firm, IOHK, is the primary developer of Cardano. The development process of Cardano has been segmented into five phases: Byron, Shelley, Goguen, Basho, and Voltaire, with each phase adding new functionality and enhancements to the platform.

The Byron phase, launched in September 2017, provided the basic foundation of the Cardano blockchain. The Shelley phase, initiated in July 2020, marked the transformation of Cardano into a decentralized network. The Goguen phase, which is currently ongoing, is added support for smart contracts and the ability to build decentralized applications on the network. The subsequent Basho and Voltaire phases aim to enhance network scalability and implement a voting and treasury system.

5.1 What makes Cardano different?

In the diverse landscape of cryptocurrencies, Cardano presents several unique technical characteristics. Each platform feature is designed to address specific problems prevalent in the current state of blockchain technology.

Two-Layered Architecture

Cardano operates on a two-layer architecture, which separates the *Cardano Settlement Layer*, the CSL, and the *Cardano Computation Layer*, the CCL. Bitcoin and Ethereum use a single-layer structure for transaction validation and computational operations, leading to potential bottlenecks and scalability issues. Cardano's two-layer approach serves to mitigate these problems.

The CSL handles transaction and token accounting, separating these basic blockchain operations from the CCL. The CCL manages smart contracts and computations. This division allows for improvements in scalability and security, as changes or upgrades to one layer do not necessarily impact the other.

Ouroboros Consensus Protocol

Cardano utilizes the *Ouroboros* protocol, a unique proof-of-stake consensus mechanism. Traditional PoW consensus protocols, like the one used by Bitcoin, consume considerable energy and face scalability limitations. Ouroboros is a response to these issues. It offers a more energy-efficient alternative to PoW protocols while still ensuring a high degree of network security. The protocol allows stakeholders with a higher proportion of *ADA*, Cardano's native cryptocurrency, to have a higher chance of being selected to add the following block to the blockchain.

Decentralized Governance

One of the significant issues facing blockchain projects is governance. Hard forks in response to disputes can lead to network splits, as witnessed with Bitcoin and Ethereum. Cardano's approach to this issue is a decentralized on-chain governance model, incorporating a treasury system and voting mechanism. ADA holders can vote on proposed changes, updates to the system, and a portion of transaction fees fund the treasury, ensuring ongoing sustainable development and reducing the risk of divisive hard forks. The voting takes place on a separate chain to reduce traffic on the mainnet.

Interoperability

An essential characteristic of Cardano is its focus on interoperability, which seeks to address the currently fragmented state of blockchain ecosystems. Many blockchain platforms operate in isolation, unable to communicate or share data. This need for interoperability presents a significant barrier to blockchain technology's widespread adoption and use.

Cardano's approach to this issue involves creating a blockchain platform that can interact with other blockchains, fostering an interconnected ecosystem. This approach involves the development of sidechains and cryptographic mechanisms that allow for secure communication between the main Cardano blockchain and other Cardano-compatible blockchains. This approach allows assets and information to be securely moved between blockchains, enhancing the functionality and utility of the Cardano ecosystem.

Native Tokens

Another distinguishing aspect of Cardano is its support for native tokens. While on some platforms, such as Ethereum, creating a new token requires deploying a smart contract, Cardano's architecture allows tokens to be created directly on the blockchain as native assets.

This feature has several advantages; native tokens on Cardano benefit from the same level of security as the ADA cryptocurrency. Transactions with native tokens are processed like ADA transactions, ensuring predictable and manageable transaction costs. Moreover, native tokens can be created, transferred, and destroyed without executing a smart contract, reducing the complexity and potential for errors.

Advanced Smart Contract Functionality

Cardano also improves upon smart contract functionality found in platforms like Ethereum. While Ethereum operates on an account-based model that can limit the parallel execution of smart contracts, Cardano employs an *extended UTXO model*, EUTXO. This model combines the security and predictability of Bitcoin's UTXO model with the ability to carry and process additional data. This model enables a greater degree of parallel execution, enhancing transaction throughput and scalability.

5.2 Layered architecture

This section discusses the difference between the settlement layer and the computation layer. The settlement layer is for accounting, and the computation layer is used for running smart contracts. These two layers are separated because transaction data is only sometimes necessary when running smart contracts and vice versa. These are two different realms within Cardano, and knowing everything can be a waste when it is unnecessary. This way of working contrasts Ethereum, where smart contracts constantly burden transactions.

Settlement Layer

The Cardano Settlement Layer is a fundamental component of the Cardano blockchain architecture. This layer is designed to track and manage the transfer of ADA.

The CSL is built on a modified version of the Unspent Transaction Output, which Bitcoin uses, known as the Extended UTXO model (EUTXO). In the UTXO model, a user does not have an account balance. Instead, their wallet software keeps track of multiple UTXOs, each representing a certain amount of cryptocurrency sent to the user's address and has yet to be spent. The user's wallet software selects one or several UTXOs as inputs to make a payment, which is then "spent" and cannot be used in future transactions.

The EUTXO model extends Bitcoin's UTXO model by allowing additional data to be carried within outputs, effectively enabling outputs to be locked by a smart contract

and facilitating more complex transactions beyond simple transfers of value. Importantly, this EUTXO model is more amenable to the parallel execution of transactions, which improves scalability. This design, coupled with the CSL's specific role of handling value transfer, isolates the settlement process from other computational functions on the Cardano blockchain.

The CSL uses the Ouroboros consensus protocol, a PoS mechanism for validating transactions and adding new blocks to the blockchain, which will be discussed in the next section. This PoS protocol allows stakeholders with more ADA to have a higher chance of being selected to add the following block to the blockchain, ensuring a more decentralized and efficient operation than the traditional Proof-of-Work mechanisms used in other platforms.

The CSL is also responsible for managing Cardano's native token ADA and any custom tokens users may create. Unlike Ethereum, where custom tokens are created through smart contracts, in Cardano, custom tokens are native and treated like ADA regarding functionality and security. This treatment of custom tokens eliminates the risk of smart contract vulnerabilities.

Computation Layer

The other layer is called the Cardano Computation Layer (CCL). As the name implies, this layer is responsible for the computational tasks of the blockchain, particularly the execution of smart contracts.

The smart contracts within the CCL are built upon Cardano's Extended Unspent Transaction Output (EUTXO) model. Cardano's smart contracts are written in a purpose-built language called Plutus, which is a statically-typed language based on the functional language Haskell. Plutus is designed to minimize potential security vulnerabilities and errors while providing powerful and expressive tools for developers to write complex smart contracts. The strong typing and functional programming paradigm can help reduce the risk of unpredictable behavior and bugs in smart contract code.

In the CCL, different users can operate under different rules when verifying transactions. For instance, a user can choose to follow the rules of a specific decentralized application and only recognize the valid transactions under those rules. This flexibility is made possible due to the distinction between the CSL and CCL. It allows various smart contract paradigms to coexist on the Cardano platform, promoting a flexible environment for decentralized application development.

Furthermore, the CCL enables users to create custom tokens, which are treated as native to the blockchain and as first-class citizens, similar to ADA, Cardano's native token. This feature provides an advantage over platforms where custom tokens are handled through smart contracts, often leading to inconsistencies and vulnerabilities.

EUTXO

The EUTXO model aims to combine Bitcoin's UTXO model with Ethereum's ability to handle smart contracts. Unlike Bitcoin, with its limited programmability, Cardano is meant to do more than only process payments. Thus, it is necessary to extend the UTXO model to allow for more programmability and expressiveness.

Ethereum's accounting model addresses this shortcoming. However, the programming semantics in Ethereum grew significantly and had unintended consequences, as discussed in the previous chapter.

Apart from the programming aspect, EUTXO needs to include two extra features

- The ability to keep the contract in its current state.
- Ensure that the same code is used throughout the transaction sequence, referred to as continuity.

Since EUTXO is deterministic and has continuity, it can also predict the transaction fees before processing. Determinism is a characteristic that account-based models lack.

Why is it called "extended" UTXO? What exactly is being extended? EUTXO extends UTXO in two ways:

- Addresses in the EUTXO model can contain arbitrary logic instead of being restricted to only public keys and signatures. Addresses are scripts. For example, when a node validates a transaction, the node determines whether or not the transaction is allowed to use a specific output as an input. The transaction will look up the script provided by the output's address and execute the script if the transaction can use the output as an input.
- Outputs can carry arbitrary data and an address and value. This data allows the script to carry state information.

The EUTXO model has several advantages over the account-based model. An EUTXO transaction is entirely deterministic; its success or failure is only determined by the transaction and its input and not by anything else on the blockchain. This results in verifying a transaction before it is transmitted onto the blockchain. EUTXO contrasts with an account-based model, in which a transaction can fail during its execution.

Transaction verification is also made easier due to the nature of UTXO since every UTXO may only be used once. The UTXO model also offers a degree of parallelism due to its local nature. When transactions operate in different branches of the UTXO graph, they do not consume the same inputs; the transactions can be processed in parallel.

In the EUTXO model, each transaction output consists of several elements:

- Value: The value in an output of the EUTXO model represents the amount of ADA or other native tokens within the output.
- Address: The address in the EUTXO model is a unique identifier for the location of ADA or other tokens within the blockchain. However, unlike traditional blockchain addresses, which are public key hashes, in the EUTXO model, the address is a composite of two parts: a datum hash and a reference to a validator script:
  - Datum Hash: The datum hash is a cryptographic hash of additional data, datum, attached to the output. It allows the validator script to access the datum, providing more context for validating transactions.
  - Validator Script: This refers to a piece of Plutus code. The validator script acts like a gatekeeper, defining the rules or conditions under which the output can be spent. This concept extends the functionality of an address beyond just receiving funds, making it an active participant in transaction validation.
- Datum: This is an additional piece of data associated with a UTXO, which can be referenced by the datum hash in the address. This datum is stored separately from the UTXO itself and is supplied by the transaction that creates the UTXO. The datum can be any arbitrary data structure, and its interpretation is up to

the validator script. This flexibility enables many use cases, including complex contracts and decentralized applications.

When a transaction attempts to spend an EUTXO, it must provide the following:

- Validator Script: this script is mentioned in the UTXO's address. It is a function in the Plutus Core language determining whether a transaction can consume the UTXO. The script takes two inputs: the redeemer and the context.
- Redeemer: the redeemer is an arbitrary piece of data the transaction provides that tries to spend a UTXO. It acts as an "argument" to the validator script, providing information or fulfilling a condition that the validator script requires to approve the transaction. For instance, in a simple payment transaction, the redeemer could be a digital signature which the validator script verifies.
- Context: the context provides additional data about the transaction trying to spend the UTXO. The context includes the transaction inputs and outputs and the validation data. The context is critical as it allows the validator script to make decisions based on the broader circumstances of the transaction, not just the redeemer.
- Datum Value: the transaction must also provide the datum value corresponding to the datum hash in the UTXO's address to spend a UTXO. This requirement ensures the validator script can access the datum during validation, enabling more complex validation logic.

The validator script, the redeemer, and the context are all passed into the Cardano script interpreter and the datum. The validator script will use these pieces of information to determine whether the UTXO can be spent.

The EUTXO model has several benefits. Including data within UTXOs allows for creation of sophisticated smart contracts, extending the functionality of UTXOs beyond simple value transfers. Additionally, by defining the conditions under which a UTXO can be spent, it is possible to enforce complex rules and protocols directly at the blockchain layer.

5.3 Consensus

Like Ethereum, Cardano also uses a proof-of-stake consensus mechanism. Naturally, some differences will be discussed in this section.

Cardano runs a consensus mechanism called Ouroboros. Ouroboros features mathematically verifiable security and is provably secure. Security, decentralization, and robustness are essential aspects of any consensus mechanism. The first difference from Ethereum is that Ouroboros uses a variant of proof-of-stake, delegated proof-of-stake. In Ethereum, everyone who wants to participate in consensus has to lock up their stake and run a validator node. However, in delegated proof-of-stake, users can pool their stakes together into a stake pool and delegate the running of a validator node to another person.

Before explaining how Ouroboros works, an explanation of delegation will be provided. Anyone who owns ADA can start delegating while retaining his spending power; ADA can be spent at any time, even while staked. Stake allows stakeholders to participate in slot leader elections in each epoch.

To start delegating, a user has to post two certificates on the blockchain: a) a staking address registration and b) a delegation certificate. The staking address differs from the usual UTXO address generated by the private keys. The different addresses mean

a regular Cardano user requires a standard payment key pair and an additional staking key pair. The staking key is then used to generate a staking reward address.

The delegation certificate includes the stake pool's verification key to indicate to which pool is being staked and the user's staking address. The verification key proves that a node can create a block. Now a transaction, the actual post, is created, which includes the delegation certificate and has as a transaction output all of the user's ada, which the user wishes to stake. The transaction output can be consumed at any time, de-delegating the ada.

With this delegation concept, any user owning ADA can allow a stake pool to generate blocks for him. The rewards paid out to stake pools and distributed over the participants are proportional to the amount of ADA delegated to the pool. However, the rewards per stake pool are subject to a threshold to centralize only some of the stakes into a few stake pools. When a pool threshold is reached, the staking pool is saturated.

To calculate how much stake is delegated to each pool, a node has to look for all transactions, which includes a delegation certificate that includes the verification key of that pool. Thus all delegated stake is public information.

Now the section continues with discussing Ouroboros. There exist several implementations of Ouroboros since it has continued to evolve. This section starts with the original implementation from 2017.

Ouroboros Classic[Kia+17]

This first implementation laid the foundations for the Ouroboros protocol. Ouroboros Classic is a proof of concept and introduces a mathematical framework to analyze proof-of-stake.

Like Ethereum's Gasper protocol, Ouroboros divided physical time into epochs. Each epoch is then further divided into slots The slots in Ouroboros are one second long. Each epoch has a different random seed to prevent pattern formation because it can also be exploited when behavior can be predicted.

For each slot, a slot leader is elected. The slot leader has the right, but not the obligation, to create a new block for this slot. The leader election process occurs at the beginning of each epoch; the period consists of a pre-determined number of slots.

First, the protocol takes a snapshot of the stake distribution at the beginning of the current epoch. This snapshot is a reference point for determining the stakeholders eligible to become slot leaders in the next epoch. Only the active stake, the tokens currently staked at the snapshot's point in time, is considered.

The slot leaders are elected from the stakeholder based on their proportional stake in the snapshot. This election uses a lottery algorithm named *Following-the-Satoshi*, FtS. This algorithm takes the set of staked ADA and divides it into its smallest denomination, a Lovelace. Then the FtS uses the epoch seed to generate a random number within the range of total Lovelace in circulation for each slot to elect the slot leader. For example: if the FtS outputs 500 for slot one, and Alice owns the 500th Lovelace. Then Alice is the slot leader for slot 1.

However, how does the network get a random seed for each epoch? The backbone of Ouroboros Classic's randomness rests on a coin-tossing algorithm. The objective is to ensure a fair, random process of selecting slot leaders for each epoch. The idea is simple: every stakeholder will use a coin-tossing algorithm to generate unbiased randomness, which can be used to generate a random string.

Once every stakeholder has generated a random string called the secret, it has to commit to its peers. Committing is required because it could change its secret depending on what other stakeholders have generated. The stakeholders send evidence to their peers to commit the secret, called a *commitment*. In the context of Ouroboros, this is essentially public-key encryption of the secret key. To reveal the secret later, the stakeholders can reveal the original secret, which their peers can verify using the commitment. One problem is that stakeholders can abort the protocol by not revealing the commitment.

Ouroboros Classic uses *Publicly Verifiable Secret Sharing*, PVSS, to solve this issue. PVSS ensures fairness in the process of seed generation. PVSS is a cryptographic method where a secret is divided into parts, giving each participant a unique part. A specific threshold number of parts is needed to reconstruct the original secret, allowing for collective ownership and recovery of the secret. PVSS also allows anyone to verify that shares have been distributed correctly without knowing the secret. It ensures that even if some participants fail to reveal their share of the secret, the original secret can still be recovered. Here the original secret of being recombined is the seed for the next epoch.

During the reveal phase, stakeholders reveal their PVSS-encrypted secrets. Any observer can then retrieve the secret shares, check the proofs, and aggregate the correctly shared secrets. The aggregated shares are then used to generate the random seed for the slot leader selection in the next epoch. This concept is where the name Ouroboros comes from; the protocol eats its tail in the previous epoch to generate a random seed for the next epoch.

This process of generating randomness is called a *Multiparty Computation*, MPC. The formal definition is a technique that allows multiple parties, each possessing fragments of private data, to participate in computing a specific result. Here this result is the random seed for the following epoch.

To make the protocol clearer, consider the following example: Suppose we have a Cardano blockchain network with ten stakeholders, labeled A through J, whom all hold an equal amount of ada, thus 10% each.

- Epoch Division: Time is divided into epochs, each comprising a certain number of slots. For simplicity, take ten slots per epoch.
- Commitment Phase: Each stakeholder generates a random value, the secret, and submits a commitment to the network. The commitment is encrypted with a publicly verifiable, secret one-time key.
- For instance, stakeholder A might generate the secret “1234”. He encrypts this value with her one-time key and sends the commitment to the network.
- Reveal Phase: After all commitments are collected and registered on the blockchain, stakeholders reveal their secrets by publishing them along with the one-time key used for encryption. The network verifies each nonce using the commitment and one-time key.
- Stakeholder A, for instance, would now publish his secret, “1234,” and his one-time key. The network can confirm that her commitment in the previous phase was indeed for the nonce “1234”.
- All revealed secrets are collected and combined into a new, single random value or seed using a deterministic function. This seed is used to select slot leaders for the upcoming epoch randomly.
- In this example, stakeholders A through J might combine the secret into the seed “5678”.

- Follow-the-Satoshi: FtS selects a slot leader for each slot in the upcoming epoch. FTS starts with the seed "5678", treats the blockchain as a list of ADA coins owned by stakeholders, and randomly picks one coin. The owner of the selected coin becomes the slot leader.
- If the first coin selected by FtS is owned by stakeholder A, then A is the slot leader for the first slot in the upcoming epoch. This process is repeated for each slot.
- Block Production: During the epoch, each slot leader validates transactions, forms a block, and adds it to the blockchain during their slot. If a slot leader misses their slot due to being offline, for example, no block is added for that slot.

Ouroboros Classic makes four assumptions:

- The network is synchronous in the sense that every stakeholder can communicate with any stakeholder within a specific time.
- There is always an honest majority of stakeholders available.
- The stakeholders do not remain offline for long periods.
- The adaptivity of malicious actors is delayed.

Ouroboros BFT[KR18]

Ouroboros Byzantine Fault Tolerance, Ouroboros BFT, is an intermediary consensus protocol introduced as a stepping stone in the transition from Ouroboros Classic to Ouroboros Praos within the Cardano blockchain. This protocol was necessary to enable the migration from the federated nodes originally used to manage the Cardano blockchain to the decentralized stake pool model implemented in Ouroboros Praos.

Key improvements introduced with Ouroboros BFT include:

- Byzantine Fault Tolerance: As the name implies, Ouroboros BFT introduces Byzantine Fault Tolerance to the Cardano blockchain. BFT means the protocol can function correctly even if some participating nodes fail or act maliciously, as long as less than one-third of the nodes are malicious.
- Transition Mechanism: Ouroboros BFT was an essential mechanism to transition from the bootstrap era, Ouroboros Classic, where a collection of trusted nodes maintained the network, to a fully decentralized network, like in Ouroboros Praos.
- Simplicity and Robustness: Ouroboros BFT is a more straightforward protocol than Ouroboros Classic or Ouroboros Praos but is also less sophisticated. It does not include the previous leaders' election mechanism. The leaders are instead scheduled ahead of time in a round-robin fashion. This change made it a good fit for the transitional phase, where robustness and reliability were critical.
- Hard Fork Compatibility: One of the critical aspects of Ouroboros BFT is that it was designed to be compatible with a hard fork. Hard forking is how the Cardano blockchain could switch from Ouroboros Classic to Ouroboros BFT without splitting into two blockchains.

Ouroboros Praos[Dav+18]

Ouroboros Praos follows a similar time structure as Ouroboros Classic, partitioning time into epochs and slots. However, it introduces a semi-synchronous network setting,

assuming an unknown upper bound on message delivery times. This assumption allows for potential network delays while assuming eventual message delivery.

In Ouroboros Praos, slot leaders are elected by each stakeholder, running a private and independent lottery for every slot. This lottery utilizes a *Verifiable Random Function*, VRF, a cryptographic primitive. A VRF maps input values to output values in a deterministic yet unpredictable manner. This mechanism allows each stakeholder to independently and privately determine their likelihood of being selected as a slot leader.

VRF takes the current epoch seed and a stakeholder's private key as inputs, generating two outputs: a random value and a proof.

The random value output of the VRF is used to determine whether a stakeholder is selected as a slot leader. Each stakeholder computes this value for each slot and compares it to a threshold proportional to the size of their stake in the cryptocurrency's total supply. If the generated value is less than this threshold, the stakeholder is designated as the slot leader for that particular slot.

The proof output from the VRF proves that a stakeholder was legitimately selected as a slot leader. When stakeholders win the leader election, they include this proof in their generated block. Other stakeholders in the network can use this proof to verify the legitimacy of the block producer without gaining knowledge of their private key. The other stakeholders use the public key to verify the proof, meaning that the random value was produced using the corresponding private key.

If a stakeholder's VRF output indicates they are a slot leader, they broadcast this output along with the VRF proof. Any other network participant can use the proof and the known public key to verify the legitimacy of the slot leader's claim without ever revealing the stakeholder's private key.

The main advantage of VRFs is that they provide publicly verifiable pseudo-randomness. In other words, VRFs allow anyone to confirm that a random output, such as the winner of a slot leader election, was randomly chosen and not fraudulently manipulated, thanks to cryptographic proof. If there are multiple slot leader candidates, the lowest VRF random value is picked as the slot leader.

Key Evolving Signatures, KES, provides an additional layer of security in Ouroboros Praos. In a KES scheme, the private signing key evolves according to a pre-determined schedule, and earlier keys are discarded. Evolving keys ensures that an adversary cannot reuse an old private key even if they manage to learn it, safeguarding against long-range attacks.

Ouroboros Praos implements a novel approach to generate the random seed for the next epoch's leader selection. The new seed is derived from the VRF outputs provided by slot leaders, ensuring that it is private, unpredictable, and not biased. This approach is critical for maintaining a fair and unbiased election process, enhancing the protocol's security.

Here is another illustrative example: Assume there is a Cardano blockchain network with four stakeholders: Alice, Bob, Charlie, and Daisy. Each holds a different amount of ada. The distribution is as follows: Alice has 40% of the total ADA, Bob has 30%, Charlie has 20%, and Daisy holds the remaining 10%.

- Epoch Division: Time is divided into epochs, each comprising several slots. For simplicity, take five slots per epoch.
- Seed Generation: At the beginning of the epoch, a seed is generated for the entire epoch. This seed is computed using the previous epoch's blocks. Let the seed for the current epoch be "5678".

- Private Slot Leader Election: Every stakeholder performs a private lottery for each slot in the epoch to determine if they are the slot leader. They input the epoch's seed and their private key into a Verifiable Random Function, VRF, which gives a random value and a proof. If the random value is less than a threshold proportional to their stake, they win the lottery and become a slot leader.
- For instance, for the first slot, Alice, Bob, Charlie, and Daisy run the VRF with the seed "5678" and their private keys. Assume the random values Alice, Bob, Charlie, and Daisy get are 0.1, 0.6, 0.3, and 0.9, respectively.
- Given Alice's stake is 40%, her threshold might be 0.4. Alice's random value, 0.1, is less than her threshold. She is elected the slot leader for the first slot.
- Bob, Charlie, and Daisy are not slot leaders for this slot, as their random values are more significant than their respective thresholds.
- Block Production: When Alice becomes a slot leader, she can create a block of transactions, sign it using her key evolving signature, include her VRF proof, and broadcast it to the network. Upon receiving the block, other stakeholders can use the included proof and Alice's public key to verify that she was a legitimate slot leader.
- Continuation of Slot Leader Election: This process continues for each slot in the epoch. The slot leader for each slot is determined privately and independently.

Through this process, Ouroboros Praos ensures that slot leader elections are random, private, and directly proportional to the stake held.

There are some subtle differences compared to Ethereum's proof-of-stake consensus mechanism. Ethereum also uses the concept of epochs and slots, but it randomly assigns a committee of validators to each slot in addition to having one leader per slot. The leader proposes a block, and the others are responsible for attesting to it.

Cardano also uses a multiparty computation protocol to select slot leaders. Each participant provides a seed to create the following random number. Ethereum, on the other hand, uses the RANDAO protocol to generate randomness.

Ethereum has a notion of finality. Each block epoch is considered final once $\frac{2}{3}$ of validators have attested to it. The block can then never be reverted to. In Cardano's Ouroboros, a block is considered final once deep enough in the chain that the probability of it being reversed is negligible. This notion of finality is the same implicit finality as in Bitcoin.

The final difference is that Cardano uses delegated proof-of-stake in contrast to Ethereum's normal proof-of-stake. People can join "unofficial" stake pools in Ethereum, but the protocol does not support them. The unofficial stake pools are programmed on the blockchain using smart contracts.

5.4 Programming

Before discussing programming on the Cardano blockchain, it is crucial to understand the EUTXO model. This subject is examined in section 5.2, but in this section, the emphasis is on how to influence the programming model of Cardano.

The validation decides whether a transaction can consume an input. In the ordinary UTXO model, this relies on digital signatures, meaning users must sign transactions for the consumption to be valid. The goal of EUTXO is to make this validation more general. It does not have just one condition, but it is replaced with arbitrary logic.

Where Bitcoin has addresses that correspond to public keys to verify a signature, there are more general addresses in Cardano. These addresses are not based on public keys or the hashes of public keys but instead contain arbitrary programs which decide the spendable conditions of UTXO.

This address requires that the input prove it can consume an output, UTXO, with a piece of data called the *redeemer*. The redeemer is used to provide the necessary information to the smart contract to validate the transaction. The redeemer can include, for example, passwords, proofs, or other forms of data required by the smart contract's conditions. The public key address is thus replaced by a script and the digital signature by a redeemer.

The next question is, what is meant by arbitrary logic?

Script Context

The first option is that the only thing a script sees is the redeemer itself, so only the logic is necessary to verify the transaction; thus, the script context consists of only the redeemer. This type of redeemer is the way Bitcoin handles this problem. As was discussed in chapter 3, Bitcoin does have smart contracts, but they are very limited. There is a locking script called simply the script on the UTXO side and an unlocking script called the redeemer on the input side. The locking script receives the unlocking script and determines if the UTXO can be consumed.

The second option is the Ethereum approach. This option is the other extreme, where the script context consists of everything, the entire state blockchain. As shown in chapter 4, Ethereum scripts are very powerful. But this is a double-edged sword, and because the scripts are so powerful, it is also difficult to predict what they do. This results in security issues and is sometimes tricky to handle for developers.

Cardano tries to find a happy medium. The script context in Cardano can see the entire transaction and not only the redeemer. It can see all the inputs and outputs of the transaction and the transaction itself. A script can then decide what to do based on this information.

The last piece of the puzzle is something called a *datum*. The datum is data that can be added to a UTXO along with the UTXO value. The datum can be thought of as the state of the script for that UTXO. This information makes it as powerful as the Ethereum model and has some advantages. Where the developer has to guess how much gas is expended by an Ethereum script, in Cardano, this can be accurately calculated due to the immutability of Cardano scripts. It is also possible to check if a Cardano transaction will validate before sending it to the chain.

The redeemer, datum, and script context are the three inputs passed to the validator when a transaction tries to spend a UTXO locked by a script. In more detail, the script context contains three pieces of information:

- The script context transaction information: information about the transaction being validated. The information includes the inputs, outputs, transferred value, and fees.
- The script context purpose: the reason why the script is being run. There are three reasons
    1. `Spending`: because the UTXO is being spent.
    2. `Minting`: a new UTXO is being created,.
    3. `Certifying`: a certificate is being validated.

Sometimes this can go wrong, for example, when a UTXO is consumed while sending the transaction. In this case, all fees are refunded. But if all the inputs are still present, one can be sure the transaction will validate.

This refund is in contrast with Ethereum. A lot can happen in the Ethereum environment between constructing the transaction and being added to the blockchain. Since Ethereum has a global state and a lot of computation happens concurrently, a lot can happen at the same time. This computation is unpredictable and can affect Ethereum scripts.

Cardano state is much more localized, which makes it easier to analyze and prove the script is valid. The scope is much more limited and easier to understand.

The spending transaction generally provides the datum, redeemer, and validator. The output at the script address only provides a hash of the script and the hash of the datum. This data means that if a transaction wants to consume a script output, the transaction has to provide the datum, redeemer, and script. Thus the user creating the transaction has to know the datum since only the hash of it is public. Knowledge is needed, but transaction output is possible by including the unhashed datum in the script. If this were not the case, then only people with knowledge of the datum from outside the blockchain could spend this output.

An example will be used to illustrate this model. Take an auction parameterized by the owner of a thing being auctioned, the thing itself, the current bid, and a deadline.

Figure 5.1 shows the first state of the auction. Alice wants to auction something and creates a UTXO at the script output. The value of the UTXO is the thing being auctioned, and the datum is empty because there is no bid yet.

Now Bob wants to bid something on the item. Bob creates a transaction with two inputs and one output. The first input is the script UTXO from the auction created by Alice. The second input is a UTXO with the value of the bid Bob wants to place, here, ten ada. The output is, again, at the output script, but with a change to the value and datum. The datum now contains Bob's address and bid. Moreover, the value contains Bob's bid of 10 ADA and the auction item.

The redeemer for the UTXO is an algebraic datatype defined by Bob called a bid. The auction script checks if all conditions are satisfied. Here the script checks if the bid happened before the deadline and whether the amount of ADA being bid is high enough. The script also checks if the transaction contains the correct inputs and outputs. The script checks whether the output contains the item and the correct datum.

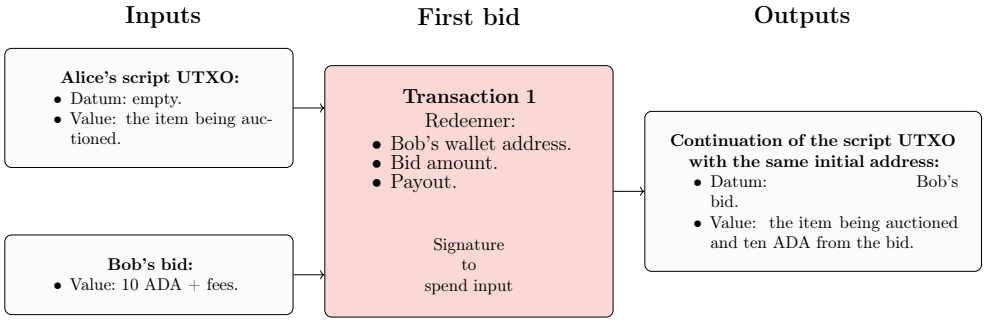


Figure 5.1: First transaction of an auction script's UTXO where a bid is placed.

Now another user, Charlie, wants to outbid Bob. He creates another transaction, as seen in figure 5.2. This transaction has two inputs and two outputs. The two inputs

in the first transaction are the bid, now 20 ada, and the auction UTXO. One output is the auction UTXO, and the second is a UTXO which returns Bob's bid.

The bid redeemer is used again. This time the script has some more parameters. The script checks that the deadline has been reached, the bid is higher than the previous bid, that the auction UTXO is correctly created, and that the previously highest bidder gets his bid back.

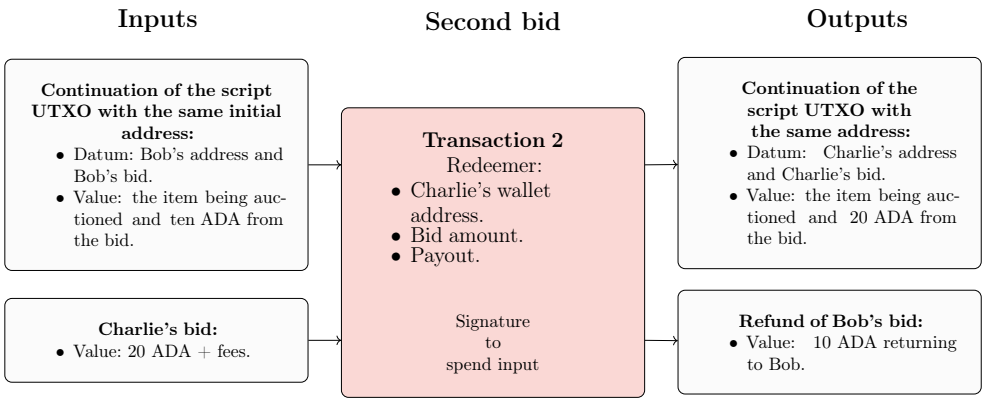


Figure 5.2: Second transaction of an auction script's UTXO where a higher bid is placed.

Now the deadline has been reached, and the auction can be closed. To close the auction, someone has to create a transaction. It could be Alice to collect her bid or Charlie who wants to collect the item. It could be anyone, but only Alice and Charlie are incentivized to do so. Figure 5.3 shows a transaction closing the auction.

The transaction has one input, the auction UTXO. This transaction uses a different redeemer, the close redeemer. It has two outputs, one for Charlie, who placed the highest bid. Charlie receives the item. The second output goes to Alice, who collects the highest bid.

When the transaction closes, the script checks that the deadline has been reached, that the highest bidder gets the item and that the owner gets the highest bid.

There is one other case. When the auction has no bidders, the close script must return the item to the owner.

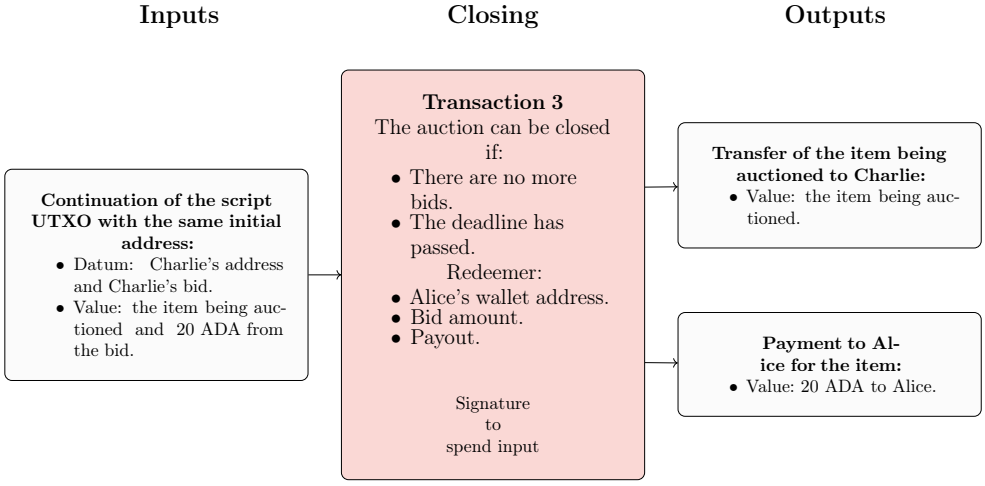


Figure 5.3: Closing of the auction once the deadline is reached.

On-Chain and Off-Chain Code

In Cardano, there is an on-chain and off-chain code.

On-chain code are the scripts that have been discussed in the previous section. These reside at the script address and are executed when a transaction tries to consume a script output. The transaction in question is only valid if the script succeeds.

When a Cardano node receives new transactions, the transactions must be validated before placing them into their mempool and block later on. The node executes each script input of the transaction, and if a script does not succeed, the transaction is invalid.

The goal of a script is thus to give a yes-or-no answer to a transaction trying to consume output.

The off-chain code's purpose is to construct transaction which passes the validation to consume UTXOs. This part of the code will not run on the blockchain but locally.

Plutus

Now that the prerequisites for programming on Cardano have been discussed, actual programming is much easier. The first platform discussed in this section is Plutus. Plutus is the primary platform to program applications on the Cardano blockchain. Plutus was released in September 2021.

When people talk about Plutus, they tend to refer to one of three things:

- Plutus Core: the low-level interpreted code that is executed by the Cardano virtual machine.
- PlutusTx: a Haskell framework that compiles to Plutus Core through the means of a GHC plugin.
- The Plutus Platform: more broadly includes Plutus Core, PlutusTx, and most tools developed around Plutus Core.

When talking about Plutus in this section, the implied meaning is PlutusTx.

PlutusTx is built as a GHC plugin. This architecture means that developers even use Haskell tooling like Cabal for it. Even so, developers are not really writing Haskell. The plugin consumes the code one writes using PlutusTx and transforms it into Untyped Plutus Core. It takes the intermediate representation of Haskell, GHC Core, and turns that into Untyped Plutus Core. This results in not needing to write a new parser and type checker. This architecture has an embedded language that looks and feels like Haskell, but the target runtime is not GHC.

PlutusTx uses advanced Haskell techniques, including Template Haskell and staged metaprogramming. These techniques allow PlutusTx to work during compile time to enable PlutusTx compilation.

The Haskell Plutus platform is notoriously difficult and frustrating to use. Even setting up a working environment can prove challenging. The difficulty is a result of including ad-hoc compiler plugins. Tools that enhance the developer experience, like language servers, usually do not work or are inconvenient to configure. The Plutus platform is tied to Haskell but then adds built-in libraries, ecosystems, and semantics, which differ from standard Haskell. All these flaws result in confusing situations when programming in PlutusTx![Ros23].

This section will nevertheless provide a brief introduction to Plutus, after which a more convenient alternative is introduced called Aiken.

Writing programming languages for Cardano is facilitated because nodes run a low-level language to validate scripts. The compilation process for PlutusTx works as follows [(]hachi1):

1. GHC: Haskell →GHC Core
2. PlutusTx compiler: GHC Core →Plutus IR
3. Plutus IR compiler: Plutus IR -> Typed Plutus Core (PLC)
4. Type eraser: Typed Plutus Core -> Untyped Plutus Core (UPLC)

Thus writing programming languages is feasible if they compile to valid UPLC. Both PLC and UPLC are stored in a binary representation on the blockchain. In its textual representation, they look like a LISP-like language. Consider the following elementary UPLC program:

```
(Program 1.0.0
  (con integer 4)
)
```

Running this program consistently results in the integer value four. Every UPLC program starts with the `program` keyword followed by a version number. The goal of this section is not to introduce UPLC, but it is nonetheless an exciting aspect of Cardano.

Template Haskell

It is interesting to understand some essential Template Haskell before starting with PlutusTx. This section will not go in-depth and assumes a basic understanding of Haskell.

In essence, Template Haskell allows programmers to manipulate Haskell code programmatically. These manipulations allow programs to:

- Generate new functions or datatypes procedurally.
- Inspect what will be generated for certain Haskell constructions.
- Execute code during compile-time.

Template Haskell uses quotes to signal a template. A Template Haskell quote is a Haskell expression `e` inside Oxford brackets: `[|| e ||]`. This quote represents the expression of the type of `e`, which is the syntax of the quoted expression.

It is possible to splice quotes into a Haskell program using the `$$`. The splice inserts the syntax of the quoted expression in the program.

A quote allows the programming to talk about Haskell programs as values. This feature can also be found in Lisp macros and is called homoiconicity. The quotes in Haskell could be more ergonomic, however.

The PlutusTx compiler can then use these quotes to compile the expression. The result of the compilation is a new quote representing the compiled Plutus Core program. The result needs to be spliced back into the main program to be used.

The same pattern is often repeated; make a quote, compile it, and then splice in the resulting compiled program.

Now some basic PlutusTx programs can be discussed.

Basic PlutusTx Programs

The following basic programs all have the boilerplate code in listing 5.1 prepended to them. This code imports the necessary libraries and enables the necessary language extensions.

```haskell
-- Necessary language extensions for the Plutus Tx compiler to work.
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell #-}

{-# OPTIONS_GHC -fplugin-opt PlutusTx.Plugin:target-version=1.0.0 #-}

module BasicPlutusTx where

import PlutusCore.Default qualified as PLC
import PlutusCore.Version (plcVersion100)
-- Main Plutus Tx module.
import PlutusTx
-- Additional support for lifting.
import PlutusTx.Lift
-- Builtin functions.
import PlutusTx.Builtins
-- The Plutus Tx Prelude, discussed further below.
import PlutusTx.Prelude
```

Listing 5.1: PlutusTx script boilerplate.

Listing 5.2 shows a basic program. The Haskell program `4` is turned into a compiled program at compile time. This program is then spliced into the Haskell program into the function `integerFour`. When inspected at runtime, by running `pretty $ getPlc integerFour`, the generated program looks like this:

```
(program 1.0.0
  (con 4)
)
```

Which is the UPLC discussed previously.

```haskell
integerFour :: CompiledCode Integer
integerFour = $$(compile [|| (4 :: Integer) ||])
```

Listing 5.2: Basic PlutusTx program which always results in the number 4 .

The identity function for integers is a slightly more complex program, as shown in listing 5.3. The result is a Plutus script that returns the same integer for a given input integer.

```
(Program 1.0.0
  (lam ds (con integer) ds)
)
```

```haskell
integerIdentity :: CompiledCode (Integer -> Integer)
integerIdentity = $$(compile [|| \(x:: Integer) -> x ||])
```

Listing 5.3: Basic PlutusTx integer identity.

The function can also be used in the compiled expression. Usually, the entire PlutusTx program is defined outside the quote and then called inside the quote. The functions need to be marked with an inlinable pragma, however. This pragma instructs GHC to keep the information about the function for the PlutusTx compiler.

The `addInteger` function is a built-in PlutusTx function that uses the integer addition from Plutus Core.

This program again results in the following UPLC program:

```
(program 1.0.0
  (con 4)
)
```

```
{-# INLINABLE plusOne #-}
plusOne :: Integer -> Integer
plusOne x = x `addInteger` 1

{-# INLINABLE myProgram #-}
fourFunction :: Integer
myProgram =
    let
        four = plusOne 3
    in four

functions :: CompiledCode Integer
functions = $$(compile [|| myProgram ||])
```

Listing 5.4: A function called within the quote.

Standard Haskell datatypes and pattern matching can also be used within the quote. These structures do not require any extra annotations.

Until now, every program has been statically defined. Sometimes programs have to be dynamically generated. For example, when creating an auction contract. The owner, item, deadline, and bids parameterize the auction.

The creation can be done in the same way as traditional functional programming. The static code is written as a function, and the arguments are provided later. There is one slight problem; the argument and function must be created at runtime. The runtime refers to the Haskell code runtime, not the Plutus Core runtime. PlutusTx solves this issue by using lifting. Lifting refers to the process of taking a function and making it work with values that are within a context.

Consider the example in listing **??**. Suppose a function, `plusOne`, operates on regular integers. It is impossible to apply it directly because `plusOne` expects an integer, not a `Maybe Int`, when using this function on a `Maybe Int`, Here is where lifting comes in. Using `fmap`, it is possible to lift `plusOne` to make it work with `Maybe Int` values, and now `plusOneLifted` works with `Maybe Int` values:

```
plusOne :: Int -> Int
plusOne x = x + 1

plusOneLifted :: Maybe Int -> Maybe Int
plusOneLifted = fmap plusOne
```

Listing 5.5: A simple lifting example.

Listing 5.6 shows lifting in PlutusTx. In this case, someone wants to apply three to the function `plusOne` at runtime, which results in a function returning four. Doing this at runtime requires the programmer to lift the argument `3` from Haskell to Plutus Core and then apply the argument. When using the lifting scheme, it is possible to run the following: `let program = getPlc $ addOneToN 4`. This results in the UPLC program below, which, when evaluated, returns four.

```
(Program 1.0.0
  [
    [
      (lam
        addInteger
        (fun (con integer) (fun (con integer) (con integer)))
        (lam ds (con integer) [ [ addInteger ds ] (con 1) ])
      )
      (lam
         arg
         (con integer)
         (lam arg (con integer) [ [ (builtin addInteger) arg ] arg ])
      )
    ]
    (con 3)
    ]
)
```

```
1  plusOne :: CompiledCode (Integer -> Integer)
2  plusOne = $$(compile [|| \(x:: Integer) -> x `addInteger` 1 ||])
3
4  plusOneToN :: Integer -> CompiledCode Integer
5  plusOneToN n =
6      addOne
7      `unsafeApplyCode`
8      liftCode liftCodeDef n
```

Listing 5.6: Lifting in PlutusTx.

A Vesting Validator Script

Now that the basics of PlutusTx have been discussed, a full validator script will be explained.

As mentioned, a validator receives information from the validating node as input arguments. This information contains the redeemer, the datum, and the script context. These three arguments are passed in as a generic datatype, called `Data` since these could be different types of values. Most of the time, developers will want to use custom data types. To achieve this, there are several template functions to convert to and from the datatypes. It is optional to write these type classes. Instead, developers can use the `unsafeFromBuiltinData` and `fromBuiltinData`. The only difference is that the former is faster but returns an error instead of a `Maybe`.

Consider the vesting example in listing 5.7. This example gifts ADA to a beneficiary, but the beneficiary can access it only after the deadline. The owner can access the funds at any time, however.

The datum requires only two pieces of information, the beneficiary address, and the deadline. Line 1 to 5 shows how the datum is created.

Next is the redeemer. The redeemer does not require any extra information since the only two pieces required to spend the UTXO are the beneficiary's signature, the owner's signature, and the current time. Both of these pieces of information are embedded in the transaction. Thus a simple `()` can be used for the redeemer, as shown in line 11.

The validator needs to check three conditions. Only the correct beneficiary or owner can spend the UTXO at the address. The signature in the transaction can be checked to validate this. The third condition is if the deadline is reached.

The public key of the beneficiary can be retrieved from the datum using `beneficiary dat`, and the owner is using `owner dat`. The public key is then passed to the `txSignedBy` function together with the transaction information. This can be seen in lines 17 to 21.

Checking the deadline is more complicated. When a validator script is run, a time check is made. The node checks that the current time falls into the valid range of the transaction before running the validator. Thus the current time lies within the validity interval.

The script context contains a time range instead of an exact time. To check the deadline, the lower bound of the interval must be later than the deadline. So, what is being checked is that the whole validity interval is to the right of the deadline. Here it is done using the `contains` function to check whether the validity interval is fully contained within the interval that starts from the deadline and extends until the end of time. This check is shown on lines 23 and 24.

Line 29 to 34 shows the validator being compiled using Template Haskell and then spliced into the program. That completes the validation logic and the section on PlutusTx.

```
1  data VestingDatum = VestingDatum
2      { beneficiary :: PubKeyHash
3      , owner :: PubKeyHash
4      , deadline :: POSIXTime
5      }
6
7  unstableMakeIsData ''VestingDatum
8
9  {-# INLINABLE mkVestingValidator #-}
10 mkVestingValidator :: VestingDatum -> () -> ScriptContext -> Bool
11 mkVestingValidator dat () ctx = traceIfFalse "beneficiary is not allowed to unlock"
       ↪ beneficiaryLock ||
12                                traceIfFalse "owner's signature is missing" signedByOwner
13   where
14     info :: TxInfo
15     info = scriptContextTxInfo ctx
16
17     signedByBeneficiary :: Bool
18     signedByBeneficiary = txSignedBy info $ beneficiary dat
19
20     signedByOwner :: Bool
21     signedByOwner = txSignedBy info $ owner dat
22
23     deadlineReached :: Bool
24     deadlineReached = contains (from $ deadline dat) $ txInfoValidRange info
25
26     beneficiaryLock :: Bool
27     beneficiaryLock = signedByBeneficiary && deadlineReached
28
29 {-# INLINABLE mkWrappedVestingValidator #-}
30 mkWrappedVestingValidator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
31 mkWrappedVestingValidator = wrapValidator mkVestingValidator
32
```

```
33  validator :: Validator
34  validator = mkValidatorScript $$(compile [|| mkWrappedVestingValidator ||])
```

Listing 5.7: Vesting validator in PlutusTx.

Aiken

As the previous section shows, development in Plutus is quite tricky. It requires detailed knowledge of EUTXO and advanced Haskell features.

Aiken tries to solve these issues. Aiken is specifically created to simplify and enhance smart contracts' development on Cardano.

Like Plutus, Aiken is still a purely functional programming language but tailored to offer developers a modern and efficient environment for constructing smart contracts on the Cardano blockchain. Aiken employs a Rust-like syntax, making it more accessible to developers familiar with Rust and similar languages than Haskell's more esoteric syntax. This syntax is meant to ease the transition and reduces the learning curve for implementing smart contracts.

However, developers must still grasp the EUTXO model to use Aiken effectively. Aiken also focuses on on-chain code but compensates by offering better compatibility with off-chain processes.

Like Plutus, Aiken also compiles to UPLC, thus offering the same advantages as Plutus while offering a more accessible development experience. Figure 5.4.

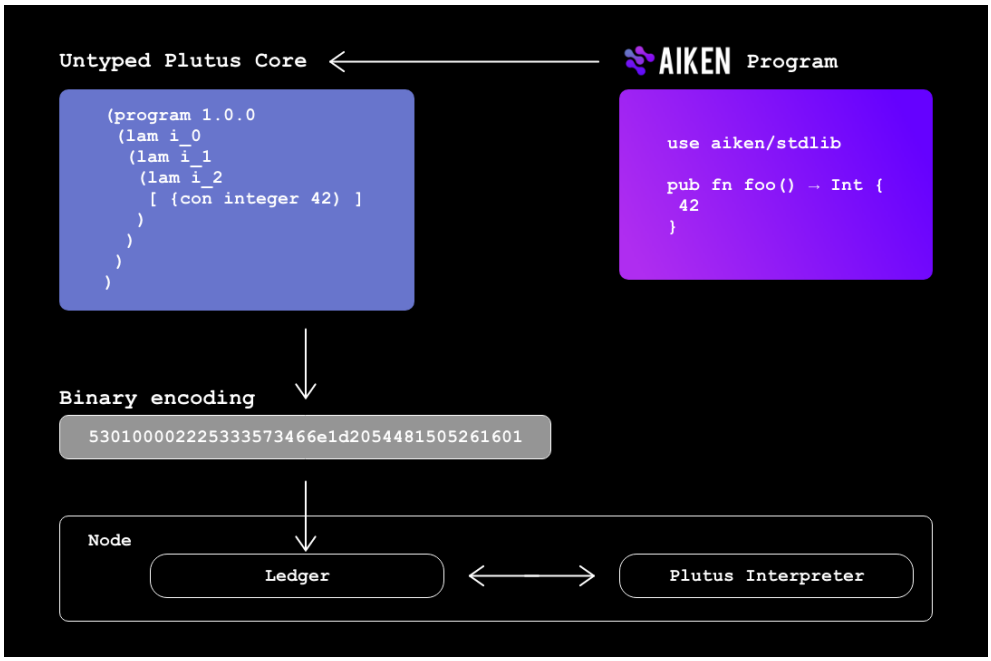


Figure 5.4: Aiken to UPLC pipeline [Ros23].

Aiken Programming Basics

The section will start with an overview of Aiken's programming basics. After the basics, some validators will be presented to show off actual programs are built.

Aiken has six basic types built into the language: booleans, integers, strings, byte arrays, data, and void. Integers and booleans are simple types in almost any language

and will not be discussed. Two building blocks for associating types are also present: lists and tuples.

Aiken supports three notations for declaring byte arrays.

1. As a list of integers ranging from 0 to 255
2. As a byte string. A byte string is how strings are represented in Aiken. The notation is simply double quotes. Text strings are only used for tracing purposes and not for running on the blockchain. Text strings are written as text surrounded by double quotes and prefixed by an `@`
3. As a hex-encoded byte string. Hexadecimal numbers are prevalent. Thus, it is logical to provide a shorthand notation. The shorthand notation is double-quotes prefixed by a `#`.

```
// As a list of bytes
// Integers can range from 0-255
#[1, 255]
#[12, 3, 55]

// As a byte string
"foo" == #[0x66, 0x6f, 0x6f] == #[102, 111, 111]

// As a hex-encoded byte string
#"666f6f" == #[0x66, 0x6f, 0x6f] == #[102, 111, 111] == "foo"
```

Tuples are used to group values together. Each element can have a different type. The notation is the following:

```
("str1", 0x55, [1]) // Type is (ByteArray, Int, List<Int>)
```

Lists are ordered collections. All the types in the list must be the same. The notation is the following:

```
[1, 2, 3, 4]     // List<Int>
[1, ..[2, 3]]    // Prepending an element: [1, 2, 3]

let x = [2, 3]
let y = [1, ..x] // [1, 2, 3]
```

`Data` is an opaque type representing any user-defined type. Data is useful when values from different types must be used in a homogeneous structure. Any user-defined type can be cast to a `Data`, and users can also convert from a `Data` type to any custom type in a safe manner. Several builtins also only work with `Data` to deal with polymorphism.

Aiken uses let-bindings for variable declarations. Values assigned to let-bindings are immutable. However, new bindings can shadow previous bindings. The let-bindings cannot be used at the top level. Thus, module constants are provided to create fixed values in the project. When such a constant is referenced, the compiler inlines the value.

```
let x: Int = 1
let y: Int = x
let x: Int = 2
let result: Bool = y + x == 3 // True

const const\_str: ByteArray = "Test string"
```

Functions are first-class citizens so that they can be assigned to variables, used as arguments, and anything else possible with any other datatype. Functions do not have an explicit return keyword but return what they evaluate. Lambda functions are also defined with similar syntax.

```
fn addOne(x: Int) -> Int {
  x + 1
}

fn twice(f: fn(t) -> t, x: t) -> t {
  f(f(x))
}


fn addTwo(x: Int) -> Int {
  twice(addOne, x)
}

fn run() {
  let addOne = fn(x) {x + 1}

  add(1)
}
```

Some functions can be promoted to be a validator. These functions have to follow certain rules:

- The functions have two or three arguments.
- The functions must be named.
- Only one or two functions may be present in the block.

Validators themselves can also take parameters. These parameters represent configurations that must be provided to create the validator instance. These parameters are then embedded into the compiled validator as part of the code. Thus, these parameters are required before the address can be computed.

```
validator(utxo_ref: ByteArray) {
  fn func1(redeemer: Data, script\_context: Data) {
    ..
  }

  fn func2(datum: Data, redeemer: Data, script\_context: Data) {
    ..
  }
}
```

Aiken also has a pipe operator. This operator provides an ergonomic way for passing the result of one function to the arguments of another. The pipe operator allows for chaining together function calls without many nesting or parentheses. The following function can be expressed with the pipe operator:

```
string_builder.to_string(string_builder.reverse(string_builder.
    from_string(string)))
// Turns into
string
  |> string_builder.from_string
  |> string_builder.reverse
  |> string_builder.to_string
```

Each line in the chain applies the function to the result of the previous line. This shorthand works nicely for function with a single argument. Functions that take more arguments will have to have specific arguments substituted.

A shorthand syntax exists for creating lambda functions that take one argument and call another function. This operation is called function capturing The underscore is used to indicate where the argument should be passed. Function capture can be used with the pipe operator to create a series of transformations on data. This usage is so common that there is a shorthand syntax for this.

```
fn add(x, y) {
  x + y
}

fn run() {
  let addOne = add(1, _)

  addOne(2)
}

fn run() {
  // This is the same as add(add(add(1, 3), 6), 9)
  1
  |> add(_, 3)
  |> add(_, 6)
  |> add(_, 9)
}

fn run() {
  // This is the same as the example above
  1
  |> add(3)
  |> add(6)
  |> add(9)
}
```

Functions with generic types are also possible to write. The variable type `T` can represent any type in the function below.

```
fn listVars(v1: T) -> List<T> {
  [v1, v1]
}
```

Aiken also has access to basic if-else control flow and pattern matching. The pattern matching is a `when expr is` expression, which says "if the data has this shape, then do that". Below is an example of matching on an integer.

```
when some_number is {
  0 -> "Zero"
  1 -> "One"
  2 -> "Two"
  n -> "Some other number" // This matches anything
}
```

The last interesting feature is the first-class support for tests. Tests can be written directly in the file with the source code and then execute the tests on-the-fly.

Simply define a test like a function, but instead of the `fn` keyword, the `test` keyword is used. The test has no arguments and returns a boolean.

Another exciting thing about tests is that they use the same virtual machine as the blockchain. Tests are thus actual snippets of on-chain code that run in the same context as production code.

```
fn addOne(inval: Int) -> Int {
  inval + 1
}

test add_test_1() {
  addOne(3) == 4
}

test add_test_2() {
  addOne(0) == 1
}
```

Moreover, the following is reported back when `aiken check` is run.

```
┍━ test1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
│ PASS [mem: 1003, CPU: 622510] add_test_1
│ PASS [mem: 1003, CPU: 622510] add_test_2
└──────────── 2 tests | 2 passed | 0 failed
Summary
  0 errors, 0 warnings
```

Basic Validators

The first validator, shown in listing 5.8, is rudimentary. Its only parameter is a number in the redeemer. Anyone who sends a transaction containing the number four in the redeemer can unlock the UTXO.

```
1  type Datum {
2    owner: Hash<Blake2b_224, VerificationKey>,
3  }
4
5  type Redeemer {
6    checkval: Int,
7  }
8
9  validator {
10     fn simple\_validator(datum _datum: Datum,
11                         redeemer: Redeemer,
12                         context _context: ScriptContext,
13   ) -> Bool {
14     addOne(3) == redeemer.checkval
15   }
16 }
```

Listing 5.8: Simple validator which checks for a number.

Now the validator can be built and submit a transaction with the validator to the testnet blockchain. To achieve this, Cardano-client will be used. Cardano-cli is the command line interface installed as part of the node and can be used to interact with the node.

The next validator does require a datum. The validator is parameterized by a verification key: the contract's owner and message. The owner part of the validator works very much like the typical non-script address; it only needs to be signed by the owner. The second requirement is that the redeemer must contain the string "Test" for the UTXO to unlock.

```
1  type Datum {
2    owner: Hash<Blake2b_224, VerificationKey>,
3  }
4
5  type Redeemer {
6    msg: ByteArray,
7  }
8
9  validator {
10   fn dat\_validator(datum: Datum,
11                    redeemer: Redeemer,
12                    context: ScriptContext) -> Bool {
13     let test\_str =
14       redeemer.msg == "Test"
15
16     let signed\_by =
17       list.has(context.transaction.extra_signatories, datum.owner)
18
19     test\_str && signed\_by
20   }
21 }
```

Listing 5.9: Simple validator with a datum and string.

Vesting Validator

The next validator is a vesting validator, just like the one described in the section on Plutus. The first five lines describe two types of aliases. One alias is for the verification key hash, and the other is for the POSIX time as an integer.

The deadline, beneficiary, and owner parameterize this validator. The datum for these parameters is shown on lines 7 to 11.

This validator also checks the purpose of the script using the script context. Line 15 shows the pattern matching done to check if the script is used to spend. When its purpose is spending, it checks if the beneficiary signed the datum and if it the time is after the deadline. The UTXO also unlocks if the owner signs the datum whenever; the deadline does not matter.

```
1  type VerificationKeyHash =
2    Hash<Blake2b_224, VerificationKey>
3
4  type posix\_time =
5    Int
6
7  type Datum {
```

```
  deadline: posix\_time,
  owner: VerificationKeyHash,
  beneficiary: VerificationKeyHash,
}

validator {
  fn vesting\_validator(datum: Datum,
                       _redeemer: Void,
                       ctx: ScriptContext) {
    when ctx.purpose is {
      Spend(_) ->
        or([signed_by(ctx.transaction, datum.owner),
            and([signed_by(ctx.transaction,
                           datum.beneficiary),
                 after(ctx.transaction.validity_range,
                       datum.deadline)])])
      _ ->
      False
    }
  }
}

fn signed_by(transaction: Transaction, vk: VerificationKeyHash) {
  list.has(transaction.extra_signatories, vk)
}

fn start_after(range: ValidityRange, lower_bound: POSIXTime) {
  when range.lower_bound.bound_type is {
    Finite(now) -> now >= lower_bound
    _ -> False
  }
}
```

Listing 5.10: A vesting validator.

As mentioned, all these validators have been tested using a locally running Cardano node connected to the test net. Cardano-cli was then used to create the transactions and test the validators.

Programming for the Cardano blockchain is more challenging than programming Ethereum. The EUTXO requires a philosophical shift. Where programming Ethereum is a lot like programming in general, the EUTXO requires a different perspective.

While programming in Plutus and Aiken takes some time to get used to, it has several upsides. It is harder to make mistakes, and when a developer makes mistakes, they get caught early before causing problems.

Aiken is especially useful for its testing capabilities. Developers can efficiently write tests for functions in the source file itself. The developer experience within Aiken is also overwhelmingly positive. The error messages are clear, and the toolchain comes with helpful developer tools, including a code formatter, language server, and testing framework.

Aiken, however, still needs to be stable and ready for general adoption. It is still in its alpha phase, and some programs might break.

Nevertheless, Aiken is a step forward for development on the Cardano blockchain.

Chapter 6

Conclusion

This thesis' aim was to explore and analyze the different aspects of blockchain technology and how these evolved over time, especially Bitcoin, Ethereum, and Cardano. The evolution of programmability of blockchains have in particular marked a step forwards in this world. Programming these decentralized systems have a allowed developers to use them for more than simply pushing currencies around.

Bitcoin, Ethereum, and Cardano all have their own unique characteristics in their transaction models, consensus protocols, and programmability.

6.1 A Rundown

Bitcoin, the pioneering system, introduced the unspent transaction output, UTXO, model, and a proof-of-work consensus protocol. Bitcoin's programmability is limited in scope, mostly being restricted to moving currency around.

Ethereum did away with the UTXO model and introduced an account-based transaction model. This model introduced more state to individual transactions since each transaction has knowledge of the entire blockchain. While this improved programmability, it did make the system less deterministic and harder to run transaction concurrently. Furthermore, each transaction changes the state of the entire Ethereum network, leading to potential scaling issues. Ethereum also shifted from a proof-of-work to a proof-of-stake consensus mechanism in its lifetime making the system more energy efficient.

Cardano saw the flaws in both system and tried to find a happy medium. Cardano is a research-first blockchain system and started out with a proof-of-stake consensus mechanism called Ouroboros. Ouroboros saw a lot of iterations over time to address different issues. Cardano also "reverted" back to the UTXO transaction model, but adding extensions to improve programmability. Cardano's extended UTXO, EUTXO, added the possibility to add additional information to every transaction while also having a special kind of transaction containing a script. This extra information is a balance between Bitcoin's model, where every transaction only has information about the unlocking script of the transaction, and Ethereum's model, where every transaction has perfect knowledge of the entire state of the blockchain. Cardano's model thus attempts to combine Bitcoin's safe and deterministic UTXO model with Ethereum's programmability.

6.2 Advancements in Programmability

Through an in-depth analysis of the different transaction models, consensus mechanisms, and programmability of the Bitcoin, Ethereum, and Cardano, the evolution of the smart contract languages can be traced.

Bitcoin's programming language, called Script, offered the first bit of programmability. Script is a non-Turing complete stack-based language. It works and feels a lot like Forth, a low-level language sometimes used on embedded devices. Its simplicity is also an advantage because it offers security. Its limited functionality means there are fewer attack vectors. But the trade-off in Script is its constrained programmability.

Ethereum meant a leap forward in programmability, which is why it is often called a "world computer". This computer run smart contracts and uses the blockchain as a means to store state. Its most common programming language is Solidity and feels a lot like a general-purpose language like C++. This high-level language allows developers to create complex smart contracts. This versatility, however, comes at a price. The Turing-complete nature of Solidity means it enables the creation of complex smart contracts, it also opens the system to security vulnerabilities. There several notable smart contract breaches are testaments to this risk. These incidents lead to financial losses, but also raised questions about the inherent security of Ethereum's concept of being a general-purpose blockchain. While Ethereum has made groundbreaking contributions to the blockchain space, especially in enabling decentralized applications, it is not without its share of challenges. These issues underline the necessity for continued advancements and improvements.

This thesis discussed two smart contract languages for Cardano. The first language is Plutus, a purely functional language embedded in Haskell. Haskell was a deliberate choice designed to make formal verification and correctness easier. Being able for verify smart contracts is an attempt to improve the security as to prevent breaches like the ones in the Ethereum system. Plutus does have disadvantages however, Haskell and how it is used in the Plutus platform creates a huge barrier to entry. Haskell is a difficult language to learn and Plutus uses advanced features, like Template Haskell, to compile the code to run on the blockchain. Besides the ergonomics of the language, the setup is also prohibitively difficult, even for experienced Haskell developers. All these flaws results in a barrier to widespread adoption.

This lead to the development of alternative languages for the Cardano blockchain. One such promising project is the Aiken language. Aiken, just like Haskell, is also a purely functional programming language which means that is has the same advantages in this area. But in contrast to Plutus, it offers a easy to setup and more developer friendly programming environment. Aiken also provides an alternative syntax compared to Haskell. Where Haskell as an ML-inspired syntax, Aiken employs a more traditional syntax similar to Rust. Besides ergonomics, Aiken also contains useful developer tools like an easy-to-use testing framework. Aiken's goal is to lower the barrier-to-entry for programming on the Cardano blockchain. This does not mean that it is easier, as developers still need a deep understanding of the EUTXO model and the properties of Cardano.

It's fascinating to observe the evolution of blockchain programming, mirroring broader trends in software development, but within a compressed timeframe. The evolution tells a story about the ongoing struggle to balance security, ease-of-use, and expressiveness. Bitcoin started with its simple language, akin to the early days of programming. Then Ethereum shifted towards a more complex, powerful, but risk-prone programming language, mirroring the rise of languages like C++ or Java. Finally, Cardano seems to embody the contemporary mindset of flexible, adaptable programming with a range of tools optimized for specific tasks. This mirror more modern programming languages

like Rust and the mindset of adopting a more functional style of programming.

6.3 Personal Reflection

Coming into this thesis I had no experience with developing or working with blockchain technologies. Understanding and evaluating how every part of a blockchain system worked took a lot longer than expected, especially since unbiased sources were difficult to find. I might have underestimated the complexity of a blockchain system.

To attempt to get a better understanding on how a blockchain system is implemented I tried my hand creating a basic system myself. I was surprised at how easy the basic datastructures and logic were to implement. The networking side of the implementation was a different story entirely. I tried to implement a realistic peer-to-peer architecture, but kept running into issues and complexities. This resulting in the implementation not being finished due to time-constraints. The effort was not wasted however, and I learned a lot about how a blockchain works.

As I have only done traditional development, the shift to blockchain programming was very weird, but also fascinating and challenging. There are no decimals, no patches, or dynamic upgrades for deployed smart contracts. In traditional programming, if a bug is found, you push a fix, and move on. In a blockchain environment, the permanence of the code and its actions requires a much higher level of diligence. Errors can lead to significant irreversible consequences, as we have seen with numerous smart contract failures. The constructor event for a smart contracts get only called once during the deployment phase and results in code having implicit owners.

The most peculiar aspect of programming on blockchain systems is that programs are essentially deployed to run on "server" supplies by other people. These people then expect to be compensated for supplying their computation power. The compensation is supplied to these people in the form of charging your programs a gas fee when the programs are executed. This fee is paid by the person initiating the transaction to your program. This model has two side effects:

- The development environment, with Ethereum I used "Foundry" and with Cardano I used a local Cardano node, will also charge the developer the same, even when the node is running on your own machine and using a testnet. This resulted in some frustration when I tried to deploy smart contracts and I did not have enough funds in my development wallet and was waiting on faucets to supply funds to continue developing.
- Code optimization does not mean improving speed or maintainability, but reducing fees when executing code. Here Cardano has a leg up because fees can be determined locally, but Ethereum has price sheets for operations. This way of thinking about code was a strange because some easy fixes have to be left in the code base to optimize gas fees.

This way of working reminds me a lot of embedded programming. In embedded programming the constraints are the hardware, but in blockchain programming the gas fees are crucial. Not only the constraints are similar, but also how to think about the platform where the program is running. In embedded programming the developer has to take into account the quirks of the hardware, like the clock speed or how the memory works. In blockchain programming, and especially the EUTXO model in Cardano, I had to take the quirks of the transaction model into account every step of the way. It really was a different way of thinking about programming and the learning curve is pretty steep.

There is one aspect I did not have the time to explore, which is formal verification of smart contracts. This looked like a very interesting subject, especially in Cardano. To

properly explore this subject I would have had to learn Liquid Haskell or a similar framework, but I did not have time to do this.

To conclude, while the transition from traditional programming to a blockchain environment comes with its own set of quirks and challenges, the experience was rewarding. As the saying goes: "A language that doesn't affect the way you think about programming, is not worth knowing". The theoretical part of comprehending the blockchain architecture took a long time, due to this I had less time to work on actual implementations. I think I have gained a deep understanding on how blockchains work and I also think this trade-off of investing more time in the theoretical side was worth it.

Bibliography

[Ant17] A.M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, 2017. ISBN: 9781491954348. URL: https://books.google.be/books?id=tponDwAAQBAJ.

[AW18] A.M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018. ISBN: 9781491971949. URL: https://books.google.be/books?id=SedSMQAACAAJ.

[Aum17] J.P. Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, 2017. ISBN: 9781593278267. URL: https://books.google.be/books?id=1D-QEAAAQBAJ.

[BA23] Demian Brener and Manuel Araoz. *OpenZeppelin*. 2023. URL: https://www.openzeppelin.com/ (visited on 05/30/2023).

[Dav+18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain." In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Cham: Springer International Publishing, 2018, pp. 66–98. ISBN: 978-3-319-78375-8.

[Dev23] Bitcoin Developer. *Bitcoin Developer Guide*. 2023. URL: https://developer.bitcoin.org/devguide/ (visited on 05/15/2023).

[Edg23] Ben Edgington. *Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond*. 2023. URL: https://eth2book.info (visited on 06/09/2023).

[Eth23] Ethereum. *Ethereum Development Documentation*. 2023. URL: https://ethereum.org/en/developers/docs/ (visited on 05/30/2023).

[Gal21] Michael B. Gale. *An Introduction to Plutus Core*. Nov. 2021. URL: https://blog.hachi.one/post/an-introduction-to-plutus-core/ (visited on 06/05/2023).

[Gre22] J. Greene. *Cardano for the Masses: A financial operating system for people who don't have one*. John Greene, 2022. URL: https://books.google.be/books?id=kHuKEAAAQBAJ.

[Hea13] Mike Hearn. *Re: 2013-03-12 IEEE Spectrum: Major Bug In The Bitcoin Software Tests The Community*. 2013. URL: https://bitcointalk.org/index.php?topic=152470.msg1620493#msg1620493 (visited on 06/09/2023).

[Int11] Transparency International. "Corruption in the Land Sector." In: *TI Working Paper* 4 (Apr. 2011). URL: https://www.fao.org/3/am943e/am943e00.pdf.

[IOG23] IOG. *Cardano Developer Portal*. 2023. URL: https://developers.cardano.org/ (visited on 05/30/2023).

[KR18] Aggelos Kiayias and Alexander Russell. *Ouroboros-BFT: A Simple Byzantine Fault Tolerant Consensus Protocol*. Cryptology ePrint Archive, Paper 2018/1049. https://eprint.iacr.org/2018/1049. 2018. URL: https://eprint.iacr.org/2018/1049.

[Kia+17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol." In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 357–388. ISBN: 978-3-319-63688-7.

[Nar+16] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016. ISBN: 9780691171692. URL: https://books.google.be/books?id=fW-YDwAAQBAJ.

[Ros23] Lucas Rose. *Aiken: the Future of Smart Contracts*. Apr. 2023. URL: https://cardanofoundation.org/en/news/aiken-the-future-of-smart-contracts (visited on 06/05/2023).

[Tea23] Foundry Team. *Foundry Book*. 2023. URL: https://book.getfoundry.sh/ (visited on 06/05/2023).

[Woo22] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger." In: (Oct. 2022). URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[Yag+18] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. *Blockchain Technology Overview*. en. Oct. 2018. DOI: https://doi.org/10.6028/NIST.IR.8202.

[Zhe+20] G. Zheng, L. Gao, L. Huang, and J. Guan. *Ethereum Smart Contract Development in Solidity*. Springer Nature Singapore, 2020. ISBN: 9789811562181. URL: https://books.google.be/books?id=OGn6DwAAQBAJ.