FACULTY OF
ENGINEERING SCIENCE

On-Chain Storage of NFT Colletions on the Ethereum Network

Axelle Moortgat

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Mens-machine communicatie

Promotor:
Prof. dr. T. Holvoet
Prof. dr. ir. B. Preneel

Evaluator:
Prof. dr. T. Van Cutsem
Frank Poncelet

Begeleider:
ir. R. Sarenche

Academiejaar 2022 – 2023

Preface

With this thesis, I end my five years of study as a computer science engineer. I would like to take this opportunity to thank everyone who was involved in this endeavor and more specifically in writing this outlined thesis.

I would like to begin to express my deepest gratitude to my promoters, Prof. Ir. Holvoet, Prof. Ir. Preneel, and Frank Poncelet. Their invaluable guidance, and profound expertise have been instrumental in shaping and guiding this thesis.

Lots of gratitude go to my supervisor Ir. Sarenche. His attention to detail, patience, and commitment were crucial throughout the entire process. His continuous dedication to reviewing and offering insightful feedback considerably improved the quality and clarity of this work.

I would like to extend my heartfelt appreciation to the individuals TokenFox and WrenCrypt who dedicated their time and expertise to answer my numerous questions. Their willingness to share their knowledge and engage in thought-provoking discussions has enriched my understanding and appreciation to the crypto community.

A special thanks goes out to my little niece who, with her creative mind, came up with the design for the OnChainBunnies. In addition, I want to thank my good friend Hannah for her inspiration and support during the entire process.

Last but certainly not least, I want to extend my deepest appreciation to my family. Their encouragement and belief in my abilities have been an immense source of inspiration and motivation.

To all those mentioned and countless others who have supported me during my studies, I offer my heartfelt thanks.

Axelle Moortgat

Contents

Abstract

The primary objective of this thesis is to provide a detailed solution for bringing an NFT collection entirely on-chain, including the images. This decreases the developer's ability to engage in malicious acts and sharing confidential data to third parties. However, bringing the collection entirely on-chain is expensive. Therefor, this thesis provides a solution to reduce deployment cost of on-chain NFT collections.

The first part of this thesis focuses on NFT collections without traits, delving into essential optimizations and exploring advanced compression techniques. The essential optimizations include injecting the data as bytes, using mappings, using memory data and optimizing the assembly code. The most significant compression factors involve rewriting paths, merging paths, removing style elements, and eliminating backgrounds. These optimizations are tested on a NFT collection of 100 images and lead to a cost reduction of 83,68%.

In the subsequent path of the research, NFT collections with traits are addressed. For this purpose, a proprietary collection called OnChainBunnies was specifically developed, comprising 5,000 distinct items. Each item within the collection represents a unique combination of eight different traits. To reduce the gas cost a solution is proposed that leverages the Loot generating algorithm, a method to assemble the images on-chain instead of storing them. The method is further optimized by using memory data, a Solidity optimizer and bitfield encoding, resulting in a 99,95% reduction in gas costs.

Finally, this thesis presents a comprehensive solution to address the presence of insider trading in NFT collections during reveal. The proposed solution for this involves the development of a sampling algorithm without replacement that selects a sample from the entire set of conceivable combinations of traits using a random number generated on-chain. Moreover, a one-to-one mapping using bitfield encoding is proposed to counteract the predictability of traits.

Samenvatting

Het primaire doel van deze thesis is om een gedetailleerde oplossing te bieden om een NFT-collectie volledig on-chain te brengen, inclusief de afbeeldingen. Dit vermindert de mogelijkheid van ontwikkelaars om kwaadwillige handelingen te verrichten en vertrouwelijke gegevens met een derde partij te delen. Het volledig on-chain brengen van de collectie is echter een dure operatie. Deze thesis biedt een oplossing om de implementatiekosten van on-chain NFT-collecties te verminderen.
Het eerste deel van deze thesis richt zich op NFT-collecties zonder traits, waarbij essentile optimalisaties worden onderzocht en geavanceerde compressietechnieken worden vergeleken en toegepast. De essentile optimalisaties omvatten het injecteren van data als bytes, het gebruik van mappings, het gebruik van memory data en het optimaliseren van de assembly code. De meest significante compressie technieken omvatten het herschrijven van paden, het samenvoegen van paden, het verwijderen van stijlelementen en het elimineren van achtergronden. Deze optimalisaties worden getest op een NFT-collectie van 100 afbeeldingen en leiden tot een kostenbesparing van 83,68%.
In het vervolg van het onderzoek worden NFT-collecties met traits behandeld. Hiervoor is een eigen collectie genaamd OnChainBunnies specifiek ontwikkeld, bestaande uit 5.000 verschillende items. Elk item in de collectie bestaat uit een unieke combinatie van acht verschillende traits. Om de gaskosten te verminderen, wordt een oplossing voorgesteld die gebruikmaakt van het Loot-generatiealgoritme, een methode om de afbeeldingen on-chain te assembleren in plaats van ze op te slaan. De methode wordt verder geoptimaliseerd door het gebruik van memory data, een Solidity optimizer en bitfield encoding, wat resulteert in een vermindering van de gaskosten voor de implementatie van 99,95%.
Tot slot presenteert deze thesis een uitgebreide oplossing om insider trading in NFT-collecties tijdens de onthulling aan te pakken. De voorgestelde oplossing hiervoor omvat de ontwikkeling van een bemonsteringsalgoritme zonder vervanging dat een steekproef selecteert uit de volledige reeks mogelijke combinaties van traits met behulp van een willekeurig gegenereerd nummer on-chain. Bovendien wordt een one-to-one mapping met behulp van bitfield encoding voorgesteld om de voorspelbaarheid van traits tegen te gaan.

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

A few years ago, crypto currencies, headlined by Bitcoin, caught on with the general public. The blockchain allowed value to be stored and exchanged digitally in a distributed manner. This promised a beckoning prospect of without central authorities such as banks. Not much later, a step further was taken with the Non-fungible tokens (NFTs) that link an account to a digital item on the blockhain. This linkage is accomplished using a smart contract. One of the main advantages of an NFT is the prove of ownership property. The blockchain makes it possible to check the originality and to track previous owners of a token. In this way, it can be proven that someone is the true owner of a particular token. Although NFTs can provide us with very powerful and unique features, the process of storing NFTs in blockchains still have some drawbacks that need to be addressed.

Even though the blockchain is touted as a trusted environment that excludes trusted third parties, this is not the case with most NFTs. A lot of trust in the developers is necessary. In August 2021, the NFT project Raccoons Secret Society launched a collection of 10 000 raccoons. The NFT collection attracted attention and achieved a trading volume of 191 ETH. A month after the launch, the developers of the collection announced that they would convert all the images of the raccoons into the image of a pile of bones. They would change not only the image, but also all metadata and rarity properties. The latter reflect the characteristics of an NFT and thus provide its uniqueness. An example of a rarirty property is the background color of the image or the fact that the character in the image is wearing sunglasses. The goal of the developers of the raccoons collection was to address the vulnerability of smart contracts [35].

Not only with the Raccoons collection did problems occur. After FTX announced their bankruptcy, FTXs domain was also unavailable. This caused all the NFTs hosted on the FTX platform to be broken. For example, the images link led to a page that gave more info about the bankruptcy. Both the image and metadata were no longer available [51].

Most NFT smart contracts only store the metadata on the blockchain. This metadata contains a link that can point to a server that stores the image or an external distributed file system like IPFS that hosts the image. In every case the link is stored in the `tokenURI` variable in the smart contract. Simply keeping the link on the blockchain has significant security implications. In fact, using the `setTokenUri()` method of the ERC721 standard, the link can be modified by the developers at a low cost [22]. Currently, there are a large number of NFT collections using this technique, even the 'blue-chip' NFT collections such as the Bored Ape Yacht Club, which has a trading volume of 677 000 ETH [47]. This leads to the customer having to trust the developer when buying an NFT, which undermines the whole concept of blockchain. Namely, this reintroduces trust in a third party.

Storing the link on-chain instead of the full image not only allows the developer to change the link but also has other major drawbacks. For example, suppose the developer decides to store the data on a server of a distributed cloud storage provider like Amazon's AWS. In this way, the NFTs are again completely centralized by entities like Amazon. This again makes the NFTs vulnerable to an attack or failure. Moreover, once the developer no longer pays to maintain this server the image is lost [41].

The reason that only the link is kept on the blockchain is economic in nature. After all, the price of keeping data on a blockchain can be high. This is because each node is going to have to keep a copy of this data. Therefore, the first goal of this thesis is to find a way to keep the image fully on-chain but still keep the cost down.

An additional trust problem lays in the fact that developers possess insider trading information. During the process of generating a collection, specific traits are assigned to each item. Consequently, each unique NFT item of the collections corresponds to multiple traits. However, when this collection is generated off-chain, the developer possesses the knowledge regarding which item results in rare traits. In 2022, Forbes published an article suspecting this kind of insider trading. Notably, a young millionaire was observed consistently bidding 32% above the average price for an NFT from the same collection, which was later discovered to have significant value. Adding to the suspicion, the young individual had shared a photo with the developer just days prior [42]. Addressing this problem necessitates the migration of an entire collection, including images, on-chain and the implementation of an algorithm that employs randomization during the reveal phase. As a result, the developer is no longer in possession of the sensitive information because it is randomly generated on the blockchain. In the last chapter of this thesis multiple on-chain sampling algorithms are discussed and compared.

1.2 Goals

The goals achieved in these thesis are two-fold. First, a solution is being explored to cheaply store images on the blockchain. Two paths are examined for this. Initially, the focus is directed towards NFT collections characterized by unique images without common traits. In this case, each image must be stored separately. In order to minimize costs, a comparative evaluation of diverse compression techniques is conducted. Furthermore, general Soldity optimizations, as well as assembly-level optimizations are examined in this section. Subsequently, attention is directed towards NFT collections wherein each item is composed of a distinctive combination of traits. An algorithm that generates the images on-chain, rather than storing them one-by-one, is examined and optimized in this part.

As a second goal, this thesis attempts to present a comprehensive solution to address the presence of insider trading in NFT collections. To counteract these insider trading problems, several sampling algorithms are considered that select a sample from the entire set of conceivable combinations of traits using a random number generated on the chain. In this way, the developer will not know which items will be selected and thus which ids will result in a rare NFT. Removing this information ensures that insider trading will no longer occur.

1.3 Research method

For each part of the study, a theoretical in-depth comparison of different solutions is proposed. To validate the theoretical findings, experimental evaluations are conducted, primarily focusing on the comparison of gas costs. This mainly compares the gas costs of the different solutions. For part one of the thesis, which pertains to the storage of an NFT collection without traits, a collection of 500 SVG images that were provided by an artist are used to conduct the several experiments. Each image posses a size ranging from 10 kB to 100 kB.

Parts two and three of this thesis concentrate on the storage of an NFT collection with distinct traits and the process of sampling items from this combination of traits, respectively. For experimental research, an in-house collection design called the OnChainBunnies was proposed. This collection contains 5000 items with a combination of eight distinct traits.

To write the smart contract, Solidity has been used. The smart contracts used in this thesis are built on the ERC721F standard. The ERC721F is an extension to the ERC721. This extension enables significant gas savings in a transaction when minting one or more NFTs. The contracts created are each tested in Remix and Hardhat. Hardhat is used for testing gas costs and general functionalities. Remix, on the other hand, is useful for testing image rendering.

1.4 Related works

The increasing success of the Ethereum blockchain also introduces increasing research into gas costs associated with deploying smart contracts. Di Sorbo et al. [34] provides an overview of common gas consuming patterns and proposing methods to circumvent such issues. Their findings are complementary to the design patterns identified by Marchesi et al. [48]. Afzal Khan et al. [43] used multiple linear regression alanysis to investigate which parameters influence high gas consumption in a smart contract. Their study corroborates the notion that the sstore opcode is a primary contributor to the elevated costs observed in most smart contracts. Furthermore, Kostamis et al. [46] investigates the differences in cost and performance for the different data locations of the Ethereum blockchain. This is also addressed in this thesis.

Moreover, within the realm of blockchain technology, a lot of research has recently been conducted into the on-chain generation of random numbers. Notably, Edginton [36] discusses the commit-reveal principle in his book, with a specific focus on the RANDAO principle implemented in Proof of Stake consensus mechanism of Ethereum. Nguyen et al. [50]elaborate on this and delve deeper into the subject and propose a commit-reveal algorithm based on homomorphic encryption that excludes the player incentive problem. In a distinct yet innovative approach, Chatterjee et al. [30] introduce a game-theoretic methodology for generating random numbers on the blockchain.

Extensive research has been conducted in the field of sampling algorithms, yielding valuable insights. Knuth's seminal work [45] offers an overview of various sampling methods, providing a foundational understanding of their principles and applications. Building upon this, Casella et al. [28] present a coherent explanation of the accept-reject method, elaborating its conceptual framework and practical implementation. Moreover, Sigman's [53] contributions shed light on the inverse transform method. An overview of different probability functions that can be used in this method is provided. However, no study into blockchain sampling algorithms has yet been conducted.

1.5 Outline

This thesis starts by providing some essential background information. In this introductory section, key concepts pertaining to blockchain technology, Non-Fungible Tokens, and distributed File Storage Systems are discussed. Building upon this foundation, Chapter 3 delves into the fundamental aspects of smart contracts for on-chain storage of NFT collections. The most important methods that will be used in a later stage are explained in detail here.

Chapter 4 focuses on the initial path pursued in this thesis, namely the on-chain storage of NFT collections without traits. This section delves into various techniques to reduce storage costs, encompassing both foundational strategies and advanced compression techniques. After storing NFT collections without traits, the focus is shitfted to collections with traits in chapter 5. The Loot algorithm is explained in combination with some optimizations.

Subsequently, in Section 6 different options to generate random numbers on the blockchain are discussed and compared in depth. The insights gained from this section form the basis for the subsequent chapter. The random number is used in the last chapter to implement a sampling algorithm to randomly select the items of the NFT collection on-chain. Chapter 7 also includes the presentation of a suitable mapping from `sampleId` to the different `traitIds` and a comparison of different sampling algorithms.

Chapter 2

Background

This section gives a brief overview of background information on blockchain technology, Non Fungible Tokens, and distributed file storage systems.

2.1 Blockchain technology

A blockchain is a digitally distributed ledger shared on a peer-to-peer network. It contains no central authority which allows transactions to be conducted without a trusted third party. A distinction should be made between public and private blockchains. A public blockchain is accessible to everyone, both for reading and writing data. Examples of public blockchains include Ethereum and Bitcoin. In a private blockchain, a central authority is responsible for granting access to the blockchain. Therefore, not everyone can view the blockchain or add transactions. An example of a private blockchain could be at a hospital where confidential patient data needs to be stored [21].

The blockchain can also be seen as a chain of blocks where each block stores a particular set of transactions. Each block then contains then transaction data, the hash of the previous block and the hash of the block itself. These hashes are responsible for linking different blocks. A hash is created by a specific hash funcion, for Ethereum this is Keccak-256 [20]. The hash of the current block is calculated from the transaction data and the hash of the previous block [44].

Adding a transaction to the blockchain requires special nodes, called miners. They are paid a certain fee for adding the block to the blockchain. The consensus algorithm decides which miner actually adds the block to the blockchain. Bitcoin utilizes the proof-of-work consensus, which involves solving a mathematical puzzle. Before publishing the mathematical puzzle, a difficulty level is calculated, which depends on the amount of collaborating mining power. This ensures that solving the puzzle always takes about the same amount of time. The node that finishes the puzzle first distributes it to the other nodes. Once the answer has been verified as being accurate, the remaining nodes validate the block. Finally, each node adds the block to the blockchain. Ethereum has been using the proof-of-work consensus algorithm for a long time but has switched to proof-of-stake since the Merge [1]. The proof-of-stake algorithm consumes much less power compared to the PoW-based consensus mechanisms. The block has the highest chance to be verified by the node that has the highest stake. A node's stake is equal to the number of coins it owns. When a node wants to participate as validator, they need to deposit a specific amount of stake on the blockchain. If the validators behave maliciously, those stakes can be destroyed. Because of this reason, it is unlikely a validator node will cheat. However, there are many other consensus algorithms, such as proof-of-importance which checks how important a certain node is for the network. Another consensus algorithm is proof of elapsed time. Each participating node in the network must wait for a certain period of time, which is selected at random. Many other consensus algorithms exist and will be invented in the future [21].

2.1.1 Ethereum

In late 2013, Vitalik Buterin, a cryptocurrency researcher, shared his white paper with the world outlining his vision for Ethereum. Two years later, the first production release followed after an online token crowdsale to fund the Ethereum project [31, 21]. The goal was to create a programmable blockchain for everyone. Ethereum is therefore an open-source and distributed platform developed by and for developers [20]. Ethereum offers many advantages over Bitcoin. While Ethereum represents a complete platform, Bitcoin represents only a cryptocurrency. Nevertheless, Bitcoin provides support for a scripting language but it falls short of Ethereum's in many areas. For example, there are poor limited functionalities which makes it difficult to write complex algorithms. In addition, transaction confirmation times are much slower than Ethereum and the language is also not Turing-complete [31, 61].

Ethereum's high-level architecture, shown in Figure 2.1, can be described as a user interacting with the Ethereum network via an Ethereum client. The Ethereum peer-to-peer network consists of two different types of nodes. First, there are the full nodes that maintain a complete copy of the blockchain. In order to set up such a node, the system must meet several requirements, e.g. at least 80 GB of free disk space [20]. However, if a system does not meet these requirements, it may choose to set up a lightweight node. Here, only the header information of the blocks is downloaded to verify that a transaction has been added to the blockchain. This verification is done using the Simple Payment Verification (SPV) method [31].

2.1.2 Smart contracts

A smart contract is a piece of software that carries out business logic that has been pre-agreed upon by two or more parties. In this way, a third party can be excluded from the transaction. A smart contract usually runs on a blockchain because of its security benefits [23]. It has a variety of characteristics, such as the fact that a smart contract is deterministic. This means that a smart contract will always deliver the

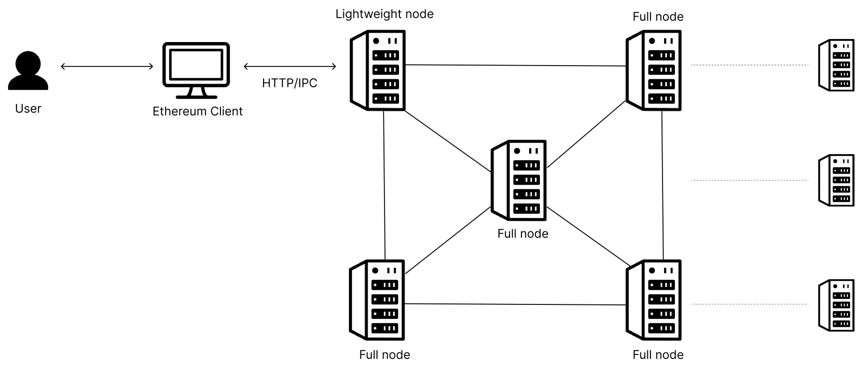


FIGURE 2.1: High-level architecture Ethereum.

same result on every node that executes it, given the state of the blockchain and the context of the transaction. Another property is the immutability of a smart contract. A smart contract's code cannot be changed once it has been deployed on the blockchain[31].

Smart contracts are usually written in high-level languages. On Ethereum, the most popular smart contract languages are Solidity and Viper. Before the contract can be deployed on the Ethereum network, it must be compiled into low-level bytecode. The compilation takes place in the Ethereum Virtual Machine (EVM). A contract creation transaction, which also generates the contract creation address, is used to deploy the contract on the network. The latter is required to identify a specific contract. A transaction that invokes this contract is necessary before it can be executed [20].

2.1.3 Transaction costs

As mentioned earlier, miners are paid a certain fee for approving and adding a transaction to the blockchain. In Ethereum, this fee is paid out in Ethereum's currency, namely ether, also identified as "ETH". Ether is a fungible token and thus can be split into several smaller units. The smallest unit is called whei and 1 quintillion whei equals 1 ether. Table 2.1 lists the different units and how they can be converted to each other [31].

To execute a transaction or a smart contract, the EVM needs fuel. Within the Ethereum blockchain, this fuel is called gas. Gas is paid in ether. Gas can be broken down into three components: gas cost, gas price, and gas limit. The gas cost is determined from the opcodes generated by the EVM. Each opcode has a particular gas cost that is specified in the Ethereum yellow paper [59]. For example, a 'sstore' opcode will have a higher gas cost than an 'add' opcode since storing data on the blockchain is more expensive than performing a one-time calculation [61]. The gas price represents the unit price of gas at that particular moment in time. It depends on how much activity there is on the Ethereum network at that time. A choice

Conversion table			
Value (wei)	Exponent	Unit	Common name
1	1	Wei	wei
1 000	10^3	Kwei	Babbage
1 000 000	10^6	Mwei	Lovelace
1 000 000 000	10^9	Gigawei	Shannon
1 000 000 000 000	10^{12}	Microether	Szabo
1 000 000 000 000 000	10^{15}	Milliether	Finney
1 000 000 000 000 000 000	10^{18}	Ether	Ether

Table 2.1: Ether denominations and conversion table

can be made to set a higher gas price so that the transaction is added faster to the blockchain by the miners. The gas limit is the upperbound of the amount of gas the user is willing to pay. If the transaction requires less gas than the gas limit specifies the excess will be refunded back to the wallet. If the transaction consumes more gas than specified in the gas limit, the transaction will fail and the gas will still be paid as a transaction fee [31].

The following formulas can be used to calculate the transaction fee [31]:

$$\begin{aligned} transactionFee_{Paid} &= Gas_{Used} * GasPrice \\ transactionFee_{Max} &= Gas_{Limit} * GasPrice \\ transactionFee_{Returned} &= Gas_{Unused} * GasPrice \\ &= transactionFee_{Max} - transactionFee_{Paid} \end{aligned}$$

The incentives behind using gas are threefold. From a financial point of view, miners are paid for adding blocks to the blockchain. This payout is based on gas. This guarantees the platform's continued existence. From a computational perspective, the halting problem is resolved by the usage of gas. By limiting gas consumption, it is feasible to know for sure that the program will stop and not become stuck in an endless loop. This clarifies the turing-completeness property of Ethereum. Finally, it might be claimed that the blockchain is vulnerable from a theoretical perspective since miners can add a block improperly. By providing miners with an economic perspective, this is combated. If the miner makes a mistake, the cryptocurrency loses all of its value [61].

2.2 Non Fungible Tokens

A Non Fungible Token (NFT) is a special type of token obtained from a smart contract. The non fungible property refers to the token not being exchangeable. This

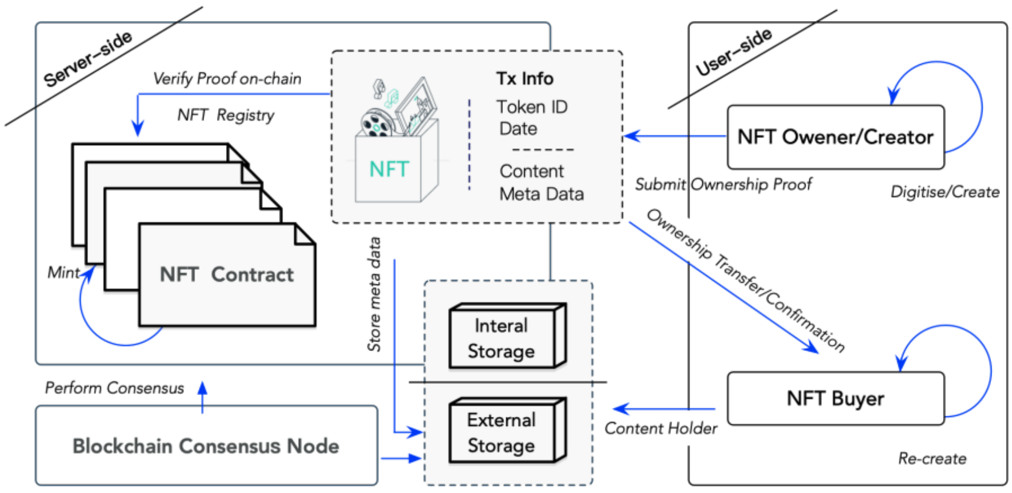


Figure 2.2: Workflow of NFTS [58].

conveys that two different NFTs are not exchangeable against each other because they are unique. Baseball cards are an example of a non-fungible object in daily life. Money, however, is interchangeable. It is tradable because each euro possesses exactly the same value and properties as others. Therefore, any two separate one-euro coins can be exchanged against each other. NFTs are therefore used to prove ownership of a digital asset. These assets include pictures, videos, music, and other things. The blockchain makes sure that it is possible to verify who held the asset at any moment, allowing for the proof of the item's existence [58, 28].

An NFT can be a specific unique artwork, such as "Everydays: The First 5000 Days" from Beeple [24] or it can be part of a collection, such as the Cryptopunks [27]. The rarity of each item in a collection is then determined by its unique characteristics. For example, only 146 of the 9998 Cryptopunks have a thick beard, which makes these items rarer and increases their value. NFTs can be purchased on various marketplaces, the most well-known being OpenSea. A creator can decide to put a certain royalty on a collection or NFT. Each time the NFT is successfully traded the given royalty fee will be paid to the creator [58].

The top-to-bottom and bottom-to-top protocols of NFT can be distinguished from one another. With the former, the NFT owner will generate the data for each image in advance; this is how CryptoPunks works. After that, the generated data is digitized and converted into a proper format. The NFT owner can choose to store the data externally or on the blockchain itself. A transaction containing the data is sent to a smart contract. The transaction is signed by the NFT owner. From now on minting and trading are possible. Minting is a special transaction where the selling address is set to 0x00000. It corresponds to an NFT buyer's initial purchase of the product [28]. Once the minting transaction has been added to the blockchain, the NFT buyer has purchased the NFT and can list it for sale again. As soon as a suitable buyer is found, the transfer transaction of the smart contract is called [62, 58].

The bottom-to-top approach allows the collection creator to create only basic rules like the characteristics of the NFT. After that, a randomized algorithm will start to produce an item using the predefined features when a buyer acquires an NFT from the collection. The minting and trading phase are analogous to the top-to-bottom approach. The general NFT workflow is illustrated in 2.2 [62, 58].

2.2.1 ERC721

The ERC721 is the token standard for Non Fungible Tokens. Token standards specify the minimum specifications for an implementation of a particular token [20]. The ERC721 specifies how to declare ownership of a token, how to create and destroy a token and finally how tokens can be transferred with authorization [61].

Each NFT has a unique `tokenId` that is saved as an `uint256` variable in the ERC721. A mapping from the address to the `tokenId` is used to determine which unique address is associated with which token. Next, the ERC721 contains a variable `metadata_url` that contains the details of the asset. This metadata also contains the link to the data or the data itself. The `metadata_url` can be retrieved using the `getTokenURI()` function. Minting the NFT is done with the public mint() function, passing the the `metadata_url` as an argument. Because each NFT is distinct, minting is done token by token, with each token being associated with the appropriate address using the `addTokenTo()` method [61, 58].

2.3 Distributed file systems

Instead of using a centralized server to store the data of the assets of the NFT, there can be opted to use a distributed file system like IPFS [5]. IPFS stands for Interplanetary File system and makes sure that a file is distributed over a different number of peers. When uploading a file to IPFS, the file will be split into chunks of 256 KB and a hash digest will be created for every chunk. Ultimately, one more separate file is created that contains the collection of the hashes of the different chunks. The hash digest of the latter file will be stored in the smart contract [46]. Because IPFS makes use of this content-based addressing there can be assured that the data is not manipulated. Modifying the data content in fact results in a different hash [40].

When the hash containing the link to the file is called via the smart contract, content discovery will take place using distributed hash tables (DHTs). These tables contain the link to the peers who host the chunks of data. The content linking of the different chunks is done using a Merkle DAG. Both the content discovery and linking take time, which make it a slower option than on-chain storage. To solve the latency of the content discovery, Pinata can be used to pin the files [7]. However, this is an expensive and sometimes complex solution [40].

Another drawback of IPFS is the fact that it is possible to keep the file distribution centralized and only the DHTs distributed. This is achieved by storing the data on one single node. When this node goes offline both the hash in the DHTs and the hash in the smart contract will remain the same. Again, all the vulnerabilities of a centralized system occur. For instance, it is almost impossible to verify the integrity of the data because there is no indication from the hash that the nodes are effectively online. [58].

IPFS was originally built as a community product and so there is a lack of strong economic incentives present. This requires nodes to host data of their own free will. Once they decide to stop hosting, the data will be lost. Filecoin offers a solution to this problem by attaching a financial reward [4]. Filecoin can be seen as a blockchain where people are rewarded with FIL tokens for hosting data. However, this still does not guarantee that the data will be kept forever. Blockchains have gone down before in the past [57].

To summarize, IPFS offers a solution against data verification. However, it is not resistant against the same centralization problems of a server. Furthermore, there should still be confidence that platforms such as IPFS or Filecoin will continue to exist and be well maintained. Finally, it should be mentioned that there are alternatives to IPFS, such as Swarn, each with its advantages and disadvantages. There are also alternatives to Filecoin such as Storj, Arweave, and Arcana Network [8, 3, 2]. However, the problem of centralization and reliance on the existence of these third parties always remains a problem [32].

2.4 Conclusion

This chapter summarized the main concepts covered in this thesis. It first discussed blockchain in general before going into more detail. The concept of Non Fungible Tokens was explained in detail. The operation of distributed file systems was discussed along with the corresponding disadvantages it offers in combination with NFTs. After reading this chapter, the reader should be fully prepared to understand the proposed solutions to store data on-chain.

Chapter 3

Smart contract Non-fungible token

Creating a smart contract for an on-chain NFT collection can easily be accomplished by using the ERC721F standard. This standard encompasses an abstract contract known as ERC721FOnChain, that can be used to implement the contract to store images on-chain. This section delves into the distinct components essential for the successful implementation of such a smart contract. The contract proposed in this section serves as the basic building block for the optimizations proposed in later sections.

3.1 ERC721FOnChain

The ERC721F standard represents an extension of the IERC721 standard, incorporating specific gas-saving techniques. This extension, developed by Frank Poncelet, is publicly available on GitHub. The ERC721F contains an abstract contract called ERC721FOnChain that serves as the basis for implementing an NFT on-chain collection [39].

Within the ERC721FOnChain abstract smart contract, four crucial methods are implemented. These methods include `getDescription()`, `tokenURI(uint256 tokenId)`, `renderTokenById(uint256 id)`, and `getTraits(uint256 id)` and can be overwritten when implementing an own version of the smart contract. The implementation of `renderTokenById(uint256 id)` is inherit from the IERC4883 interface [18]. This interface is responsible for handling the concatenation of SVG images, ensuring their correct rendering.

3.2 Contract implementation

This thesis start with presenting a simple smart contract, extending the ERC721F-OnChain, that enables minting of an SVG image in order to compare the various techniques. There are four distinct functions in this smart contract. The `flipSaleState()` function makes it possible to open up the minting phase. The implementation is illustrated in Listing 3.1 [39]. The quantity of tokens the buyer wants to mint is passed as a parameter to the `mint()` method, which is seen in Listing 3.4 [39]. The function will fist check if the requested number of tokens is available and that the limit has not been reached. Once everything these checks succeed, the amount of tokens will be minted by calling the `_mint()` function of the ERC721F standard [39].

Listing 3.1: FlipSaleState method in Smart Contract

```
function flipSaleState() external onlyOwner {
        saleIsActive = !saleIsActive;
    }
```

The `tokenURI()` method will be crucial for storing images and metadata on the blockchain itself. It is not implemnted in the contract but get inherited from the ER721FOnChain. Listing 3.2 shows how the method is put into practice [39]. A base64 encoded JSON string that contains the token's metadata will be returned. An example of a tokenURI is shown in Figure 3.1. Every tokenURI can be distinguished by the beginning with '`data:application/json;base64,`'. Base64 encoding is used when binary data must be transported across different mediums so no data can get lost. In this case, the data is transported from the smart contract to the web browser. Web browsers are therefore capable of decoding this base64 data. After decoding the JSON string, the metadata can be distinguished. An example of a decode `tokenURI` is shown in Figure 3.2. This metadata consists of the name, description, image and the attributes of the NFT [22].

Listing 3.2: tokenURI method in Smart Contract

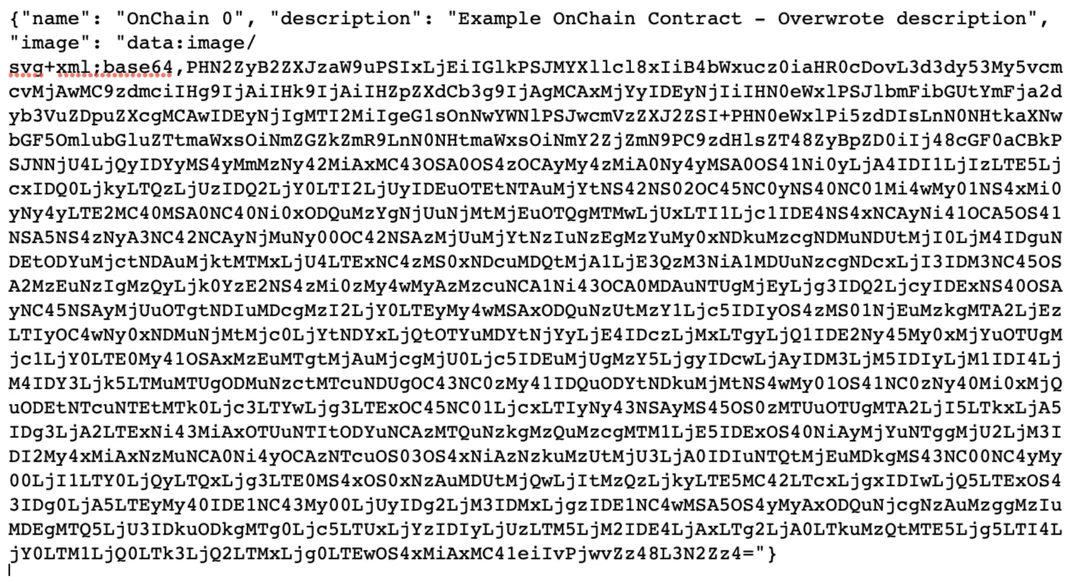
```
function tokenURI(uint256 tokenId)
        public
        view
        override
        returns (string memory)
    {
        require(_exists(tokenId), "Non-Existing token");
        string memory svgData = renderTokenById(tokenId);
        string memory traits = getTraits(tokenId);
        string memory json = Base64.encode(
            bytes(
                string(
                    abi.encodePacked(
                        '{"name": "',
                        name(),
                        " ",
                        Strings.toString(tokenId),
                        '", "description": "',
                        getDescription(),
                        '", "image": "data:image/svg+xml;base64,',
                        Base64.encode(bytes(svgData)),
```

```
{
        "0": "string: data:application/
json;base64,eyJuYW1lIjogIk9uQ2hhaW4gMCIsICJkZXNjcmlwdGlvbiI6ICJFeGFtcGxlIE9uQ2hhaW4gQ29udHJhY3QgLSBPdmVyd3JvdGU
gZGVzY3JpcHRpb24iLCAiaW1hZ2UiOiAiZGF0YTppbWFnZS9zdmcreG1sO2Jhc2U2NCxQSE4yWnlCMlpYSnphVzl1UFNJeExqRWlJR2x
rUFNKTVlYbGxjbDh4SWlCNGJXeHVjejBpYUhSMGNEb3ZMM2QzZHk1M015NXZjbWN2TWpBd01DOXpkbWNpSUhnOUlqQWlJSGs
5SWpBaUlIWnBaWGRDYjNnOUlqQWdNQ0F4TWpZeUlERXlOaklpSUhOMGVXeGxQU0psYm1GaWJHVXRZbUZqYTJkeWIzVnVaRH
B1WlhjZ01DQXdJREV5TmpJZ01USTJNaUlnZUcxc09uTndZV05sUFNKd2NtVnpaWEoyWlNJK1BITjBlV3hsUGk1emRESXNMbk4wTkh0
a2FYTndiR0Y1T21sdWJHbHVaVHRtYVd4c09pTm1aR1prWm1SOUxuTjBOSHRtYVd4c09pTm1ZMlpqWm1OOVBDOXpkSGxzWlQ0OF
p5QnBaRDBpSWo0OGNHRjBhQ0JrUFNKTk5qVTRMalF5SURZeU1TNHlNbU16Tnk0Mk1pQXhNQzQzT1NBME9TNHpPQ0F5TXk0ek1
pQTBOeTR5TVNBME9TNDFOaTB5TGpBNElESTFMakl6TFRFNUxqY3hJRFEwTGpreUxUUXpMalV6SURRMkxqWTBMVEkyTGpVeU
lERXVPVEV0TlRBdU1qWXROUzQyTlMwMk9DNDVOQzB5TlM0ME5DMDFNaTR3TXkwMU5TNHhNaTB5Tnk0eUxURTJNQzQwTV
NBME5DNDBOaTB4T0RRdU16WWdOalV1TmpNdE1qRXVPVFFnTVRNd0xqVXhMVEkxTGpjMUlERTROUzR4TkNBeU5pNDFPQ0E1
T1M0MU5TQTVOUzR6TnlBM05DNDJOQ0F5TmpNdU55MDBPQzQyTlNBek1qVXVNall0TnpJdU56RWdNell1TXkweE5Ea3VNemNnTk
RNdU5EVXRNakkwTGpNNElEZ3VOREV0T0RZdU1qY3ROREF1TWprdE1UTXhMalU0TFRFeE5DNHpNUzB4TkRjdU1EUXRNakExTG
pFM1F6TTNOaUExTURVdU56Y2dORGN4TGpJM0lETTNOQzQ1T1NBMk16RXVOeklnTXpReUxqazBZekUyTlM0ek1pMHpNeTR3TXl
Bek16Y3VOQ0ExTmk0M09DQTBNREF1TlRVZ01qRXlMamczSURRMkxqY3lJREV4TlM0ME9TQXlOQzQ1TlNBeU1qVXVPVGd0TkRJ
dU1EY2dNekkyTGpZMExURXlNeTR3TVNBeE9EUXVOelV0TXpZMUxqYzVJREl5T1M0ek1TMDFOakV1TXprZ01UQTJMakV6TFRJeU
9DNHdOeTB4TkRNdU5qTXRNamMwTGpZdE5EWXhMalF0T1RZdU1EWXROall5TGpFNElEY3pMak14TFRneUxqUTFJREUyTnk0NU1
5MHhNall1T1RVZ01qYzFMalkwTFRFME15NDFPU0F4TXpFdU1UZ3RNakF1TWpjZ01qVTBMamM1SURFdU1qVWdOelk1TGpneUlEY
3dMakF5SURNM0xqTTVJREl5TGpNMUlESTRMak00SURZM0xqazVMVE11TVRVZ09ETXVOemN0TVRjdU5EVWdPQzQzTkMwek15
NDFJRFF1T0RZdE5Ea3VNak10TlM0d015MDFPUzQxTkMwek55NDBNaTB4TWpRdU9ERXROVGN1TlRFdE1UazBMamMzTFRZd0xqZ
zNMVEV4T0M0NU5DMDFMamN4TFRJeU55NDNOU0F5TVM0NU9TMHpNVFV1T1RVZ01UQTJMakk1TFRReExqQTVJRGczTGpBMk
xURXhOaTQzTWlBeE9UVXVOVEl0T0RZdU5DQXpNVFF1TnprZ016UXVNemNnTVRNMUxqRTVJREV4T1M0ME5pQXlNall1TlRnZ01
qVTJMak0zSURJMk15NHhNaUF4TnpNdU5DQTBOaTR5TVRNeUlEZ3NRZFJhLjMxMDIuMTk1TUNMTDMTX0TVNakEwSURJdU5U
UXRNakV1TURrZ01TNDNOQzAwTkM0eU15MDBMakkxTFRFZ ...
```

FIGURE 3.1: TokenURI of NFT

```
                                bytes(traits).length == 0 ? '"' : '", "attributes": ',
                                traits,
                                "}"
                            )
                        )
                    )
                );
        return string(abi.encodePacked("data:application/json;base64,", json));
    }
```

The `renderTokenById()` method, illustrated in Listing 3.3, is called within the `tokenURI()` method [39]. This enables the display of a certain SVG image on a screen. The SVG data is initially saved as a string in this procedure. The string is then ABI-encoded. The Application Binaray Interface (ABI) is the interface that represents interaction between contracts in the Ethereum ecosystem. ABI encoding will then return a JSON format that contains the base64 encoded imagedata, distinguished by the beginning '`data:image/svg+xml;base64`'. This is illustrated in Figure 3.2 A browser can interpret this and allow it to render the image. So a web browser like Opensea that wants to display the image will always render the image in real time instead of fetching the image from a server or IPFS and caching it. The fact that the image object in the JSON string is a base64 encoding of the image rather than a link to a server or IPFS is crucial in this case [61, 22].



```
{"name": "OnChain 0", "description": "Example OnChain Contract - Overwrote description",
"image": "data:image/
svg+xml;base64,PHN2ZyB2ZXJzaW9uPSIxLjEiIGlkPSJMYXllcl8xIiB4bWxucz0iaHR0cDovL3d3dy53My5vcm
cvMjAwMC9zdmciIHg9IjAiIHk9IjAiIHZpZXdCb3g9IjAgMCAxMjYyIDEyNjIiIHN0eWxlPSJlbmFibGUtYmFja2d
yb3VuZDpuZXcgMCAwIDEyNjIgMTI2MiIgeG1sOnNwYWNlPSJwcmVzZXJ2ZSI+PHN0eWxlPi5zdDIsLnN0NHtkaXNw
bGF5OmlubGluZTtmaWxsOiNmZGZkZmR9LnN0NHtmaWxsOiNmY2ZjZmN9PC9zdHlsZT48ZyBpZD0iIj48cGF0aCBkP
SJNNjU4LjQyIDYyMS4yMmMzNy42MiAxMC43OSA0OS4zOCAyMy4zMiA0Ny4yMSA0OS41Ni0yLjA4IDI1LjIzLTE5Lj
cxIDQ0LjkyLTQzLjUzIDQ2LjY0LTI2LjUyIDEuOTEtNTAuMjYtNS42NS02OC45NC0yNS40NC01Mi4wMy01NS4xMi0
yNy4yLTE2MC40MSA0NC40Ni0xODQuMzYgNjUuNjMtMjEuOTQgMTMwLjUxLTI1Ljc1IDE4NS4xNCAyNi41OCA5OS41
NSA5NS4zNyA3NC42NCAyNjMuNy00OC42NSAzMjUuMjYtNzIuNzEgMzYuMy0xNDkuMzcgNDMuNDUtMjI0LjM4IDguN
DEtODYuMjctNDAuMjktMTMxLjU4LTExNC4zMS0xNDcuMDQtMjA1LjE3QzM3NiA1MDUuNzcgNDcxLjI3IDM3NC45OS
A2MzEuNzIgMzQyLjk0YzE2NS4zMi0zMy4wMyAzMzcuNCA1Ni43OCA0MDAuNTUgMjEyLjg3IDQ2LjcyIDExNS40OSA
yNC45NSAyMjUuOTgtNDIuMDcgMzI2LjY0LTEyMy4wMSAxODQuNzUtMzY1LjcxIDIyOS4zMS01NjEuMzkgMTA2LjEz
LTIyOC4wNy0xNDMuNjMtMjc0LjYtNDYxLjQtOTYuMDYtNjYyLjE4IDczLjMxLTgyLjQ1IDE2Ny45My0xMjYuOTUgM
jc1LjY0LTE0My41OSAxMzEuMTgtMjAuMjcgMjU0Ljc5IDEuMjUgMzY5LjgyIDcwLjAyIDM3LjM5IDIyLjM1IDI4Lj
M4IDY3Ljk5LTMuMTUgODMuNzctMTcuNDUgOC43NC0zMy41IDQuODYtNDkuMjMtNS4wMy01OS41NC0zNy40Mi0xMjQ
uODEtNTcuNTEtMTk0Ljc3LTYwLjg3LTExOC45NC01LjcxLTIyNy43NSAyMS45OS0zMTUuOTUgMTA2LjI5LTkxLjA5
IDg3LjA2LTExNi43MiAxOTUuNTItODYuNCAzMTQuNzkgMzQuMzcgMTM1LjE5IDExOS40NiAyMjYuNTggMjU2LjM3I
DI2My4xMiAxNzMuNCA0Ni4yOCAzNTcuOS03OS4xNiAzNzkuMzUtMjU3LjA0IDIuNTQtMjEuMDkgMS43NC00NC4yMy
00LjI1LTY0LjQyLTQxLjg3LTE0MS4xOS0xNzAuMDUtMjQwLjItMzQzLjkyLTE5MC42LTcxLjgxIDIwLjQ5LTExOS4
3IDg0LjA5LTEyMy40IDE1NC43My00LjUyIDg2LjM3IDMxLjgzIDE1NC4wMSA5OS4yMyAxODQuNjcgNzAuMzggMzIu
MDEgMTQ5LjU3IDkuODkgMTg0Ljc5LTUxLjYzIDIyLjUzLTM5LjM2IDE4LjAxLTg2LjA0LTkuMzQtMTE5Ljg5LTI4L
jY0LTM1LjQ0LTk3LjQ2LTMxLjg0LTEwOS4xMiAxMC41eiIvPjwvZz48L3N2Zz4="}
```

Figure 3.2: TokenURI of NFT decoded

Listing 3.3: renderTokenById method in Smart Contract

```
function renderTokenById(uint256 id)
        public
        view
        override
        returns (string memory)
    {
        require(_exists(id), "Non-Existing token");
        string memory part;

        part = '<svg version="1.1" ... </svg>';
        return
            string(
                abi.encodePacked(
                    part
                )
            );
    }
```

A notable distinction between the implementation of an on-chain NFT collection and an off-chain NFT collection resides in the implementation of the `tokenURI` method. Specifically, in the case of an on-chain NFT collection, the `tokenURI` method encompasses both the metadata and the complete SVG string associated with the NFT. Conversely, in the context of an off-chain NFT collection, the `tokenURI` method incorporates the metadata along with a hyperlink referencing the image. This hyperlink can direct to a server or utilize a decentralized file storage system like IPFS.

Listing 3.4: Mint method in Smart Contract

```
function mint(uint256 numberOfTokens) external {
        require(msg.sender == tx.origin, "No Contracts allowed.");
        require(saleIsActive, "Sale NOT active yet");
        require(numberOfTokens != 0, "numberOfNfts cannot be 0");
        require(
            numberOfTokens < MAX_PURCHASE,
            "Can only mint 30 tokens at a time"
        );
        uint256 supply = totalSupply();
        require(
            supply + numberOfTokens <= MAX_TOKENS,
            "Purchase would exceed max supply of Tokens"
        );

        for (uint256 i; i < numberOfTokens; ) {
            _mint(msg.sender, supply + i);
            unchecked {
                i++;
            }
        }
    }
```

3.3 Conclusion

This thesis leverages the ERC721FOnChain abstract smart contract as a foundational framework for the implementation of an on-chain NFT collection. The implementation of the `tokenURI` method is of great importance here. In contrast to off-chain collections, the entire SVG image is stored as a string and utilized in this method. The assembling of the SVG image is achieved through the `renderTokenById` method. The initial contract presented in this section serves as a starting point for further development and exploration in this thesis.

Chapter 4

NFT Collections Without Traits

This thesis will examine two paths to successfully store images on the Ethereum network at the lowest cost. The first path focuses on NFT collections with absence of traits, which necessitate individual on-chain storage. In this regard, this section explores multiple standard Solidity optimizations. Furthermore, this solidity code is translated into opcodes by the Ethereum Virtual Machine (EVM). This step in the process can also be optimized. The second subsection delves into proposed solutions and strategies within this optimization realm. Finally, 100 SVG images are injected with the listed optimizations into a smart contract as a test. To cut the expense of gas, these images are compressed using a variety of approaches. There are two phases to these compression techniques. In the first phase, techniques are applied using a tool. Phase two involves some manual application of procedures.

4.1 Scalable Vector Graphics

Scalabe Vector Graphics, or SVG for short, is a vector file format for displaying two-dimensional graphics. The file format dates back to the 1990s but only became popular around 2017 due to further developments of the Web. The reasons behind the popularity of SVGs in the web are two-fold. For one, it is a vector file type which ensures that when scaling the image, the resolution will not increase or decrease. On the other hand, SVGs are written in XML code. By storing the image as text information, search engines can read and use the keywords [37].

4.2 Solidity optimizations

4.2.1 Data locations

In Solidity, three different types of memory locations are provided, namely storage, memory and calldata. The first one is used to hold state variables, which implies that all functions in the smart contract can access them. Each blockchain node also keeps a copy of these variables. As a result, data storage is the most expensive type of data. Memory data can be thought of as temporary data. They are utilized in the

Table 4.1: Gas costs storage data vs memory data

	Storage data [gas]	Memory data [gas]	reduction [%]
Contract deployment	4 360 174	3 632 576	16.69

logic of functions or as function parameters. This data will be less expensive because it is relinquished after the function call. Calldata is similar to memory data in the sense that it is temporary but has some other properties. For instance, calldata must be used as a dynamic parameter of an external function, is immutable, and can not be used in the logic of a function [61, 17].

It is also crucial to keep in mind that the three various data storage have different behavior in terms of assignment. For example, an assignment from memory to memory data or from storage to a local storage variable will result in a reference. All the other types of assignments will create a copy of the corresponding variable[17].

In an experiment, a 2362-byte image was deployed using the standard contract from Chapter 3. First this image was stored in the contract as storage data, then it was stored as memory data. The results of the experiment are shown in Table 4.1. Using memory data instead of storage data reduced the gas price for deployment by 16.69 %. Since call data may only be utilized as a function argument and not in function logic, it was not possible to compare call data in this experiment.

4.2.2 Injecting data

Each transaction, and thus the deployment of a smart contract, is limited by a certain amount of gas. Since the London upgrade, this amount, known as the block size, equals 30 million gas [10]. Therefore, it will not be possible to deploy the smart contract with all the data added as memory or storage data in advance. In fact, the deployment cost will exceed 30 million gas. In this case, it may be opted to inject the images with separate transaction after the deployment of the smart contract. A mapping from an index to the SVG data as storage data will then be included in the smart contract. With a separate function that takes the index and the SVG data as parameters the key-value pair is added to the mapping with an individual transaction [49].

Problems may arise when implementing the injection method. This can lead to an issue where Remix, the development environment, fails to recognize the string format. As a result, injecting SVG data as a string object is not possible. To address this problem, a potential solution involves pre-converting the SVG data into an alternative data format prior to injection. Since the image data is converted to base64 in the `tokenURI` method, this pre-conversion step can be carried out off-chain and injected

TABLE 4.2: Gas costs injection base64 vs byte array

	base64 [gas]	byte array [gas]	reduction [%]
Transaction cost	2 640 443	1 987 603	24.72

into the smart contract in this format. Another approach entails converting the SVG data into a byte array and subsequently injecting it. This is also a conversion that takes place in the `tokenURI` method.

In an experiment, both approaches were used with the 2362-byte image and their gas costs were evaluated. The results are shown in Table 4.2, which demonstrates that byte injection is the best option. Listing 4.1 contains the corresponding code for this function.

LISTING 4.1: Injecting SVG data

```
function addImage(uint256 index, string memory svgData) external
   onlyOwner{
        parts[index] = svgData;
    }
```

4.2.3 Array vs Mapping

It can be opted to store the injected data in an array or a mapping. The latter always transfers the token's ID to the proper bytearray. This is because, as shown in Figure 4.2, the structure of an array in Solidity resembles a mapping together with the length of the mapping. As a proof of experiment, three images of respectively, 2 362 , 18 986 and 19 003 bytes were injected to a mapping and an Array. The results, illustrated in 4.3, show that a mapping is the ideal choice. From now on, all the coming experiments will use the mapping structure.

LISTING 4.2: Structure of Array in Solidity

```
struct Array{
    mapping(uint => bytes) items:
    uint lenght;
}
```

4.3 Optimizing assembly

Before the smart contract can be deployed on the Ethereum network, the Solidity code will be converted into opcodes by the Ethereum Virtual Machine (EVM). This translation can be made more efficient. For this, the Solidity compiler has an integrated optimizer. Both the opcode-based optimizer and the Yul optimizer

Table 4.3: Gas costs mapping vs listing

	Cost mapping [gas]	Cost list [gas]	reduction
Deployment cost	3 691 060	3 695 916	0,13 %
Inject image 1	1 728 350	1 750 145	1,25 %
Inject image 2	13 539 125	13 543 808	0,03 %
Inject image 3	13 402 995	13 407 678	0,03 %

will be active if this optimizer is enabled in Solidity. The former will apply some simplification rules for opcodes and combine equal code. Finally, it also gets rid of unnecessary code. The Yul-based optimizer will carry out several optimization stages that lead to shorter or even more optimized code [6].

The built-in optimization requires a parameter to be included, namely the "Paramater Runs". This parameter is an assumption of how many times an opcode will be run. For instance, if there are 500 NFTs in the collection, the mint function will be called 500 times, making the number of parameter runs in that scenario 500. The parameter can be thought of as a trade-off between the deployer's and the buyer's costs. As an illustration, a 'Parameter Run' of 1 will result in a low deployment cost but a high function execution cost [6].

As an experiment, the described optimizer was applied and compared to a non-optimized smart contract. Several experiments were conducted with increasing 'Parameter Run' of 1, 50, 200 and 400 runs. All the smart contracts used in this experiment contained the compressed test image of 2 362 bytes. As shown in Table 4.4, the contract deployment cost increases with the number of runs. In contrast, the minting cost will decrease with increasing number of runs. An optimizer with a 'Parameter Run' of 1 will, therefore, result in the best reduction for the contract deployment. This results in a reduction of 39.54%. With a reduction of 1.02%, a parameter run of 400 for the mint function is ideal. A "Parameter Run" of 1 may appear to be ideal because it produces the greatest reduction, but this is not always the case. After all, unlike the mint cost, the deployment cost is an expense that is incurred only once. However, the mint expense will be incurred 500 times for a collection of 500 NFTs. Therefore, there is a trade-off that needs to be made.

4.4 Compressing techniques

SVG data can be compressed using a variety of compression methods. Here it must be taken into account that the quality does not decrease. Compression takes place off-chain because of gas costs. The compressed data will then be used in the smart contract and this will provide a reduced value for deployment. Utilizing this technique does not affect the gas cost of the mint function. The latter is a fee that the customer, not the deployer, must pay. It is crucial to note that these compression

Table 4.4: Gas cost built-in optimzer Solidity

Deployment/ Method	Cost non-optimized [gas]	Cost 1 run [gas]	Cost 50 runs [gas]	Cost 200 runs [gas]	Cost 400 runs [gas]	Reduction
Contract deployment	3 878 649	2 345 072	2 347 160	2 395 607	2 442 554	39.54 %
Mint()	96 047	95 425	95 128	95 071	95 071	1.02 %

strategies will only lower deployer costs, not buyer costs.

A set of 100 images between 0.5 and 10 kB will be subjected to a series of compression algorithms. After that, an average will be computed to show the effect of the overall compression. An average percentage will be used to show the contribution to the total compression for each compression technique separately.

4.4.1 Big compressions

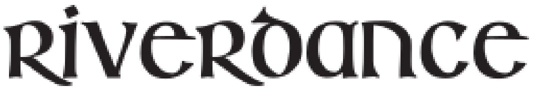
One of the compression techniques with the greatest impact is path rewriting. This optimization technique will remove unnecessary anchor points and round off the remaining anchor points but is classified als a noisy compressing technique. However because of the potential for visual distortion, caution is required in this situation. This method was implemented using the SVGO v3.0.0 tool [16]. Here, it is possible to specify the number precision. This parameter specifies how many anchor points are traced. By increasing the precision, the number of anchor points will also increase and thus the quality will decrease less. This approach offers the biggest size decrease, namely 70.51%, on average across all photos. It must be emphasized, however, that low number precision is consistently chosen. As seen in Figure 4.1 on the left in comparisons to right, this results in a drop in quality.

The second largest compression technique used was path merge. Often several paths are used in SVG code, each denoted by the '<path/>' element. These paths can be merged so that the '<path/>' string can be removed from the image multiple times. This results in an average compression of 15.32%. This method was implemented using the SVGO v3.0.0 tool [16] but can however, also be performed quiet easily manually.

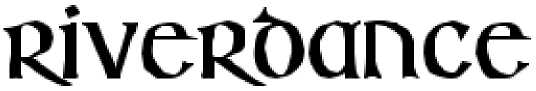
Finally, removing the style elements also resulted in an average reduction of 12.14%. However, this again must be approached with great care. Removing the style elements can have major consequences, one such example is illustrated in Figure 4.2. This method was implemented using the SVGO v3.0.0 tool [16].

Table 4.5: Compressing techniques with reduction contribution

Compressing Technique	Reduction gas cost
Round/Rewrite paths	70.51 %
Merge paths	15.32 %
Remove style elements	12.14 %

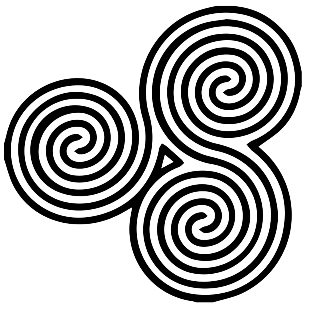


(a) Image before compression

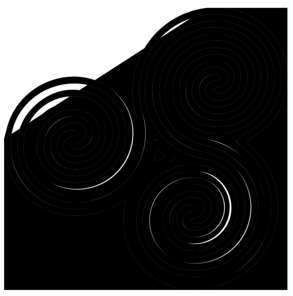


(b) Image after compression

Figure 4.1: Noisy compression with path rewriting



(a) Image before style removed



(b) Image after style removed

Figure 4.2: Removing style element

4.4.2 Small compressing techniques

Quite a few other compression techniques were used using the SVGO v3.0.0 tool but each resulted in an average compression of less than 1%. These compression techniques involve clean up of whitespaces and enters, collapsing useless groups and removing unnecessary elements. Examples of these unnecessary elements are: enters, comments, metadata element, unused defs, unused namespaces, etc [16].

It should be noted that the XML instructions could also be removed as part of the compression process. Yet here, caution is required. This is because rendering an SVG image without these instructions is not possible in all web browsers. It is

Compressing phases	Total size [kB]	Total transaction cost [gas]	Size reduction compared to previous phase	Gas compared to from previous phase
Non-compressed images	2 332 496	1 658 730 063	/	/
Compressed images	545 941	391 462 729	76.59 %	76.40 %
Compressed-by-hand images	368 644	270 747 197	32.48 %	30.84 %

TABLE 4.6: Size and transaction cost reduction of the three compressing phases

advisable to maintain these instructions in the code for easy handling.

4.4.3 Manual compressing techniques

After applying well known compression techniques, some techniques were also applied by removing non-essential code. Here the biggest reduction was the elimination of white backgrounds. In fact, often a path element was used to fill the white surfaces, these paths can obviously be removed. Furthermore, classes, defs and groups were removed. The total average size reduction of these compressing techniques compared to the compressed image from the previous section is 32.48%, as can be seen in Table 4.6.

4.4.4 Transaction costs compressing techniques

One by one, the 100 SVG pictures were injected into the smart contract using the aforementioned method after the two phases of compression were performed. The summary figures are displayed in Table 4.6. This demonstrates that using the SVGO v3.0.0 compression methods results in a average cost reduction of 76.40%, or even a reduction by a factor of 4.24. Given these compressed images, the additional manual compressions produced a further decrease of 30.84%, or a factor of 1.48, in transaction cost. Together, all of the compressions result in an average cost reduction of 83.68 %, or a compression by a factor of 6.13.

The average reduction in transaction costs should be noted to be smaller than the average reduction in image sizes. This is the case because Ethereum uses a fixed 32-byte storage slot. If there is any space left, this slot will always be filled with 0-bit values. As a result, each image's final slot is further padded with 0 bits, which results in storage loss.

4.5 Conclusion

This section described a method for adding existing images to the smart contract on-chain. Injection of the SVG file as a string, Base64 encoding, or binary data were contrasted. In terms of transaction cost, the later choice produced the best outcomes. Additionally, several data locations were evaluated, with storage data emerging as the top option. Furthermore, the utilization of an array rather than a mapping was analyzed, with a mapping emerging as the least expensive solution. Furthermore, the application of a Solidity optimizer with a parameter run of 200 led to significant reductions in contract deployment (39.54%) and minting (1.02%) costs.
Finally, various compression methods were performed. Notably, the highest size reductions were found to be achieved by rounding or rewriting paths, merging paths, removing style elements, and removing backgrounds. These compression procedures resulted in a cost decreases of 70.51 , 15.32 , 12.14 , and 30.84 %, respectively. The compression procedures resulted in a decrease of a factor of 6.13 or an average price reduction of 83.68%.

Chapter 5

Storing NFT Collections with Traits

The second part of this thesis focuses on NFT collections that incorporate traits. By storing only the traits within the smart contract, as opposed to the complete images, the gas cost for storage can be significantly reduced. The actual image of the NFT is then constructed on-chain by combining the stored traits. The implementation of this method is discussed in detail using a self generated collection, further referred as the OnChainBunnies. The OnChainBunnies collection comprises 5000 items, each possessing eight unique traits. In a first attempt, the logic of the Loot smart contract is applied. The smart contract will then be gradually optimized.

5.1 On-chain generation

The OnChainMonkeys are the greatest illustration of a collection that has been successful in creating an on-chain collection with various attributes. There are 9500 distinct items in this collection, each with six unique features and a different attribute for the background color. It is not possible to count the latter as unique. Together, the six separate features add up to 163 different values, which might be assembled into 164 341 716 unique items [14]. Nevertheless, it is worth noting that the OnChainMonkeys collection was not the original pioneer in generating images on-chain. In reality, they adopted and replicated the method utilized by the Loot collection [11].

This thesis presents the OnChainBunnies (OCB), a self-created collection of 5000 items, to apply the method that was utilized with the Loot collection. Each of the following traits are present in each NFT: bracelet, eyebrows, hat, glasses, tie, purse, wizard, and whiskers, with corresponding possible values of 6, 4, 30, 25, 28, 8, 5, and 4. Thus, a total of 80 640 000 unique combinations are possible. The bunnies also have a background trait but this one is not counted for uniqueness.

Each bunny in the smart contract is represented by a distinct `sampleId` that is algorithmic transformed into a structure made up of the various `traitIds`. The `sampleIds` represent all the possible unique values there exist in the collection. This means that, in the case of the OCB example, the `sampleIds` range from 0 to 80 639 999. to In listing 5.1, this is shown. Later chapters of this thesis go into extensive detail about the algorithmic conversion.

Listing 5.1: random number generator OnChainBirds

```
struct Bunnie {
        uint8 background;
        uint8 bracelet;
        uint8 eyebrows;
        uint8 hat;
        uint8 glasses;
        uint8 ties;
        uint8 purse;
        uint8 magic;
        uint8 whiskers;
    }
```

The Loot algorithm's gas-saving trick is in the way the images are stored in the smart contract. This means that just the attributes are stored as storage data in the smart contract as opposed to each image individually. Following the retrieval of the `tokenURI`, the image is generated on-chain and returned to the caller. `renderTokenById(uint256 id)` is called by the function `getTokenURI( uint256 id)`. The latter performs the assembly. The implementation of it is shown in Listing 5.2. As an illustration, this listing also demonstrates how the hat traits are stored before being assembled in the `getHat(bunnie.hat)` method. A hat consists of an upper part that can consist of five different colors and a lower part that can consist of five different colors. The colors come in pairs each time. This makes it possible to have 3x3x5=30 different hats. `hatId` zero results in an image without a hat. Similar methods are used for constructing the other attributes. A fully assembled bunny is illustrated in Figure 5.1. The cost to deploy the full contract with an optimization of 200 runs results in a gas price of 13 207 163. This is comparable to the contract to inject images one at a time that was covered in Section 5. This injection method for Figure 5.1 costs 2 703 077 gas. For a full collection of 5000 items, this would result in a cost of approximately 13 515 385 000 gas. Application of the Loot algorithm thus results in a reduction by a factor of 866.

Listing 5.2: random number generator OnChainBirds

```
string[] private hat_up = ['<rect y="205" x="475" width="130"
    height="60"   stroke-width="8" fill="#',
        '<rect y="140" x="485" width="110" height="120" stroke-
           width="8" fill="#',
        '<circle cy="238" cx="540" stroke-width="10" r="62" fill
           ="#'];
string[] private hat_down = ['<rect y="262" x="440" width="200"
              height="40" stroke-width="8"  fill="#',
        '<ellipse ry="22" rx="70" cy="280" cx="540" stroke-width
           ="8" fill="#'];
```

```
string[] private hat_colors_up = ['5fbf00', '000000','e960ff', '56
    aaff',    '5fbf00'];
string[] private hat_colors_down = ['56aaff','ff0000','ff0000', '
    ffff93',   'ffff93'];

function getHat(uint8 hat_id) public view returns(string memory){
        if (hat_id == 0){
            return '';}

        return string(abi.encodePacked(
            hat_up[hat_id/5%3],
            hat_colors_up[hat_id%5],
            hat_basic,
            hat_down[hat_id/15%2],
            hat_colors_down[hat_id%5],
            hat_basic)
        );
    }

function renderTokenById(uint256 id)
        public
        view
        override
        returns (string memory)
    {
        require(_exists(id), "Non-Existing token");
        Bunnie memory bunnie = randomOne(id);

        string memory output = string(abi.encodePacked(frame[0],
            background[bunnie.background], frame[1], frame[2],
            getGlasses(bunnie.glasses)));
        output = string(abi.encodePacked(output,
        getHat(bunnie.hat),
        getEyeBrows(bunnie.eyebrows)));
        output = string(abi.encodePacked(output,frame[3],
        getBracelet(bunnie.bracelet),
        getTie(bunnie.ties),
        getMagic(bunnie.magic),
        getPurse(bunnie.purse),
        getWhiskers(bunnie.whiskers),
        frame[4]));
        return string(output);
    }
```

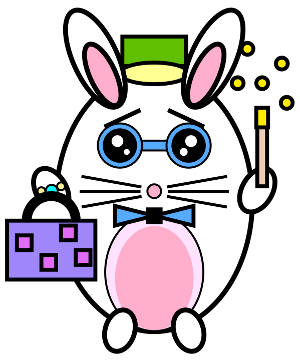


Figure 5.1: Assembled bunny

5.2 Opimizations on the Loot method

5.2.1 Assembly optimizer

The standard Solidity optimizer, as explained in Section 4 can be used as an initial optimization. Nevertheless, this raises a new issue. The optimizer converts every argument supplied to a method from call data to data on the stack, which is a significantly less expensive data location. The maximum depth of this stack is 2024 slots, each of which contains 256 bits [23]. Due to this, the function `abi.encodePacked()` raises an error when concatenating the different strings for the immage assembly. By completing the concatenation in several steps, this problem is resolved. The function `renderTokenbyId(uint256 id)` in listing 5.2 also provides an illustration of this separation of concatenations. Table 5.1 shows the results of applying the assembly optimizer on the smart contract. It was decided to set the number of runs to 200 to have a healthy division of gas costs between developer and buyer. It can be concluded that using the assembly optimizer can lead to a lower gas cost of 14.9 % or a reducing factor of 1.18.

5.2.2 Data locations

All of the SVG components are maintained as storage data in the Loot smart contract, which is the most costly data location. Therefore, as a performance improvement, one may choose to move the arrays of SVG strings to the related get method as memory data. However, the array that contains the data from the bunny's frame cannot be moved to the `renderTokenById(uint256 id)` since then the contract creation initialization data would return with length of more than 24576 bytes. These bytes represent the contract code size limitation introduced in the Spurious Dragon fork [60]. The arrays containing the weights of the traits cannot be converted to memory

TABLE 5.1: Results optimizations in terms of gas cost

Technique	Deployment cost [gas]	Optimization factor	Optimization percentage
One-by-one storing	~13 515 385 000	~855	~99.88%
Loot generation without optimizer	15 806 809	1.17	14.90%
Optimizer with 200 runs	13 452 033	1.17	14.90%
Memory data	7 490 396	1.80	44.32%

data since memory data can never be dynamically. However, the `usew(uint8[] memory w, uint256 i)` method only works with dynamic arrays. After applying these optimizations, the deployment cost equals 7 396 816 gas units. This is reduction of 44.32% in comparison with the previous optimization. The results are shown in 5.1.

5.3 Conclusion

The Loot generation process was used as the foundation for the low-cost method of storing images on-chain [11]. Generating the image instead of saving the full image ensures that the gas price can be divided by an average factor of 855. There are three sequential methods to improve the Loot contract. As the first method, the Solidity optimizer can be used as an initial optimisation. This leads to a lower gas cost of 14.9 %. Second, all of the SVG components are moved to memmory data. This optimization introduces a reduction of 44.32 % in comparison with the previous optimization.

Chapter 6

Random Number Generation on the Blockchain

NFT collections in which the images consist of a composition of traits are often assembled in a random manner. The distribution of traits that results from this random process implies that some NFTs are more valuable than others because of their rarity. One or more random numbers may be used to generate all of the NFTs. For example, one can opt to choose a random item from all available sampleIds each time. Another option is to select combination of random traits for eacht item. Yet it is not as simple as it seems to generate this random number on the blockchain. This section outlines the problem of generation of random numbers. Four different types of solutions are presented, each having pros and cons.

6.1 The problem of generating random numbers

The Ethereum blockchain's deterministic nature makes it difficult to generate random numbers. Every node must reach the same consensus as a result of this determinism. A random number, however, cannot be predicted, hence the node cannot know in advance what should happen when a random number is generated. This leads to a pardox. Therefore, there is a lack of probabilistic nature in smart contracts [30].

For many blockchain applications, this probability is necessary. It might be necessary to implement a lottery on the blockchain, for instance. Also, the gaming business uses random numbers frequently. Probability also comes into play when analyzing an NFT collection in which the attributes are assembled in a completely random manner.

In the context of purchasing an NFT collection, it is common for buyers to be unaware of the specific NFT they will acquire. In this scenario, the NFT launch occurs in stages. During the initial minting phase, customers have the opportunity to purchase one or more NFTs from the collection. At this stage, all NFTs in the collection are identical. The actual image and traits associated with each NFT are not revealed until a later stage known as the reveal phase. Hence, purchasing an

NFT from a collection at the start of the process is comparable to gambling on a valuable image from the collection and to purchasing Pokemon or baseball cards. Therefor, the integrity of the randomness is vital to prevent any potential insider knowledge from being exploited.

6.2 Random seed generation with block information

6.2.1 Block variables

Using the data from blocks is a first approach to on-chain random number generation. This block's information is subject to frequent, unpredictable adjustments. The random number is then generated using the block information as a seed. Block variables can be viewed as the first category of block data. The timestamp, address of the miner, gas limit, or difficulty of a block are a few examples of block variables. Nonetheless, it is clear how a miner may alter this data. For example, a miner can wait a few seconds to validate the block, causing the timestamp to turn into a favorable seed for him [52]. It should also be noted that since Ethereum's merge, the difficulty of a block turns out to be zero or prevrandao value. This is because the Proof-Of-Stake consensus mechanism replaced the Proof-Of-Work consensus mechanism, eliminating the block difficulty in the consensus process. Therefore, using block difficulty is not a good way to generate a random number on the Ethereum blockchain [36]. Finally, the miner incentive problem, which will be covered in more detail later in this chapter, makes the use of block variables inadvisable.

6.2.2 Blockhashes

Block information is handled more securely when block hashes rather than block variables are used. This is due to the miner's inability to modify the hash. Yet, there are still some issues and insecurities with this approach. Three different forms of block hashes are conceivable: the hash of the previous block, the current block and the future block. This section outlines all three of them.

Blockhash of previous block

The hash of the previous block can be retrieved in solidity with the following code: `hash(block.number-1)`. The seed that was used to create the random number is then matched up with this hash. Then, the random number can be created by, for instance, converting the keccack hash to an integer. The keccack hash offers a uniform distribution in this process. The use of this approach has two drawbacks. On the one hand, the blockhash is predictable during the brief interval between the validation of the previous and current blocks. It is impossible to rule out an attack during this time. On the other hand, this approach results in the miner incentive problem [52].

A scenario where a lottery is organized and a miner participates in this lottery

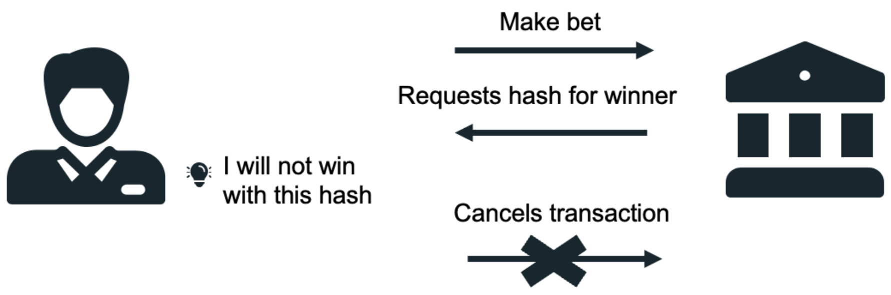


Figure 6.1: The miner incentive problem

as a participant in order to highlight the miner incentive problem is proposed. Figure 6.1 illustrates the entire procedure. The player, who in this instance also operates as the miner, first places a bet with the house. After the house receives the bet, it asks the hash of the previous block in order to determine the winner. The miner notices that the requested hash will yield a negative result, implying that he will not earn the prize. As a result, the miner cancels the transaction and pays a penalty. However, the miner has provided an additional possibility of winning the lottery by forcing the house to complete another transaction in order to acquire the hash [54, 30].

As long as the fee for canceling the transaction is less than the lottery prize, the miner will always cancel the transaction in his favor. In some occasions, BTC Relay is utilized to solve this issue. The block hashes in this technique are derived from the Bitcoin blockchain rather than the Ethereum blockchain. This is due to the higher cost of canceling a transaction on Bitcoin. But, in most circumstances, the cancel charge will be less than the price gained, and hence will not resolve the issue [54, 30].

Blockhash of future block

To address the miner incentive problem, the future blockhash principle can be used. The entire procedure is shown in Figure 6.2. First, the player will send a transaction to the house containing a particular bet. The house will then store the block number of the player's transaction on the blockchain. The player asks the house to reveal the winner during the second stage of this process. The house will retrieve the appropriate block number from the blockchain in order to respond to this query. The winner of the lottery is determined by the hash of this block number. The miner can no longer affect the request because it depends on the player's transaction, which was committed in the past. This method resolves the miner incentive problem, nevertheless it raises a new security concern. The Ethereum network only stores hashes of the most recent 256 blocks due to scalability reasons. So, if a player waits until more than 256 blocks have been confirmed after his bet, the block number's hash will equal zero. The winning number may then be deduced from this [52]. This hack took place in the SmartBillions lottery and resulted in a loss of 400 ETH [26].

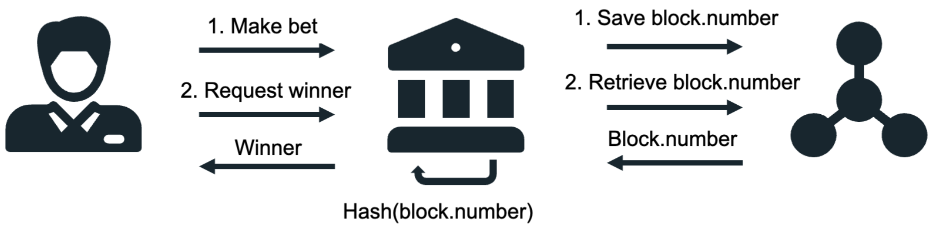


Figure 6.2: Principle behind using a future block hash

Blockhash of current block

Finally, many smart contracts also use the current blockhash. This is to make an attempt to eliminate the miner incentive problem. However, the blockchash of the current block is not yet known and thus will result in zero. Needless to say, this is not a good method.

6.3 Oracles

Using an oracle is a second kind of approach for creating random numbers on the blockchain. An oracle is a mechanism for bringing off-chain data onto the blockchain. This can be required, for instance, to know the current dollar value of a specific cryptocurrency or to use meteorological data in a smart contract. However, these oracles are helpful for more than just putting the data onto the blockchain. Oracles can also be used to get a random number, which was generated off-chain, on-chain [9].

Oracles come in two different forms. There are centralized oracles on the one hand, and decentralized oracles on the other hand. In the former case, a provider takes care of the delivery of the data. Yet this creates a single point of failure and undermines confidence, which makes it not a suitable option. In a decentralized version, the oracle is made up of a network of nodes that independently provide the needed information after reaching a consensus. However, it can happen that the requested information comes only from one data provider and all these nodes address the same data provider. Once again, there is a single point of failure, undermining trust. Chainlink is a good illustration of a decentralized oracle on the Ethereum network.The architecture of Chainlink is illustrated in Figure 6.3 [9].

Chainlink has introduced a Verifiable random function, or VRF, to undermine trust in the external data supplier. Smart contracts can then ask the oracle nodes for random numbers using the VRF principle. Every node is in the possession of a private and public key. An unpredictable seed is sent to the nodes whenever a smart contract requests a random number. This seed can be the block hash of the block which contains the `requestRandomness` call and other specific parameters. The node will then produce a random number using the provided seed and its private key. By comparing the hash with the public key and at the seed, a proof of correctness

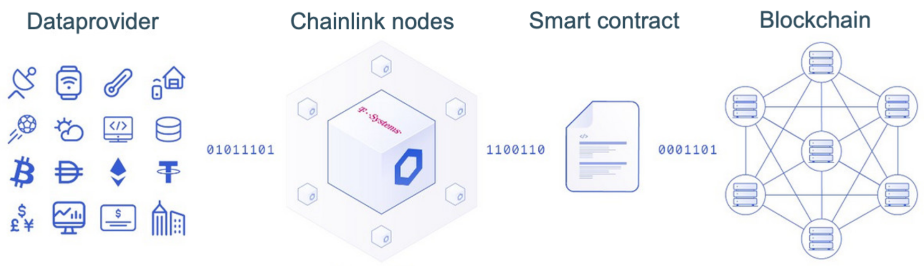


FIGURE 6.3: Architecture of Chainlink

can then be produced [25, 19].

Based on security, choosing an oracle is not always the best option. In fact, oracles are particularly prone to front running attacks. Figure 6.4 illustrates this attack. The house, in charge of selecting a lottery winner at random will, first ask the oracle for a random number. The oracle subsequently sends a transaction back to the house along with the requested random number. The write-back transaction ends up in the transaction pool. However, there exists an attacker that catches sight of this transaction and submits a bet with a larger gas price. The hacker's bet will be included in a block before the write-back request due to the increased gas price. As a result, the house will receive the attacker's bet prior to the specified random number. The lottery's eventual winner will be revealed to be the attacker [30].

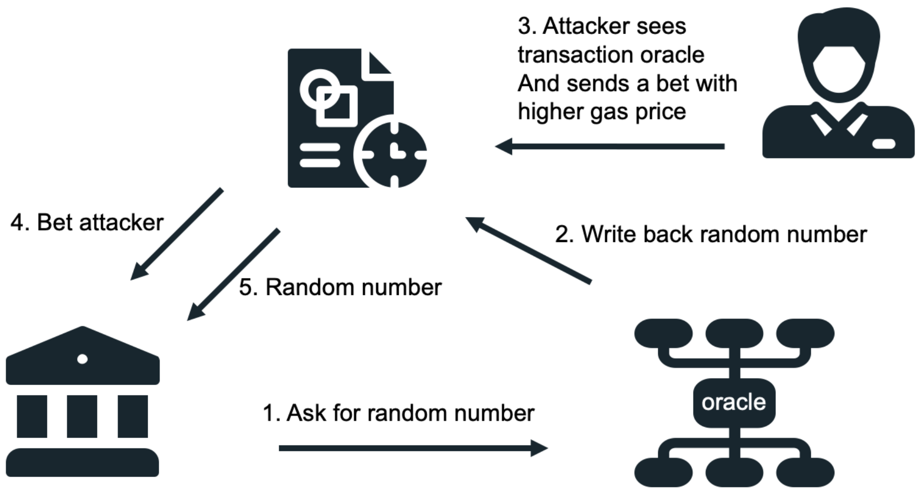


FIGURE 6.4: Front running attack oracle

6.4 Commit-reveal

A third type of random number generation consists of the commit-reveal principle. There are two distinct steps to this approach. In Figure 6.5, phase one is represented in red, and phase two in blue. Initially, a number of voluntarily participating individuals independently choose a random number outside of the blockchain. After that, they individually submit the hash of these random numbers to the house. Phase two begins once all hashes have been delivered to the house. In this phase, each player sends their previously chosen random number to the house. The house verifies from the hashes that these numbers correspond to their initial choice of number. Finally, if this is the case, the house will add up the numbers and use this sum as the random number [50].

This technique also possesses some drawbacks. First, the player incentive problem has replaced the miner incentive problem. In the case of the player incentive problem, the voluntary participant is also a lottery player. The final participant, who is also a player, is able to check which number the previous participants have sent in stage two. As a result, the player will not send his number if adding his number results in a sum for which he does not win the lottery. This starts over the entire procedure, giving the player a second chance to win. A second drawback is the interaction required when implementing this system. The commit-reveal method requires voluntary participants each time when you require a random number. Using RANDAO or Quanta might be a solution to this. These libraries uses the commit-reveal concept, with volunteers receiving a reward each time they pass a random number. Nevertheless, doing so necessitates putting confidence in the RANDAO and Quanta developers [30]. A final drawback is that the volunteer is not rewarded by forwarding an effective random number. Hence a volunteer can always forward zero without any consequences. However, it suffices that one of the participants sends a random number. The outcome in this scenario will always be random [50].

There are numerous techniques to address the player incentive problem. Using the blockhash together with the sum is the simplest option. Both the volunteer participant and the miner are unable to forecast the blockhash or the random number, respectively [30]. Using homomorphic encryption is another option. The following equation serves as the foundation for the homomorphic encryption property [50]:

$$E_k(a) + E_k(b) = E_k(a + b)$$

This means that the sum of the encryption of a number a and the encryption of a number b amounts to the encryption of the sum of and a and b. El-Gamal encryption is an example of encryption mechanism that possesses the homomorphic property. In this scenario, each participant sends their random number to the house encoded with El-Gamal. The house can then take the sum of these encrypted numbers and decrypt it. By doing this, the player incentive problem is resolved and fewer transactions must be issued overall. Moreover, there is just one decryption as opposed to three [50].

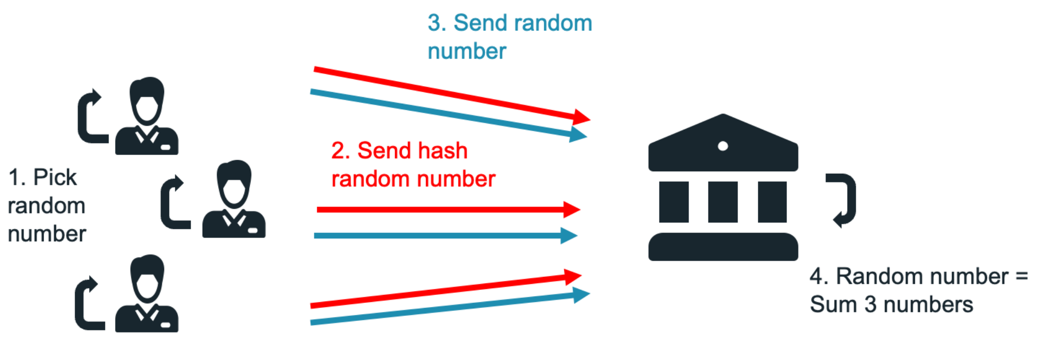


FIGURE 6.5: Commit-reveal principle

6.5 Prevrandao opcode

Since the Ethereum merge, the block.difficulty opcode has been replaced by a new value known as block.prevrandao. The prevrandao number is generated using the RANDAO library and is utilized for random assignment of the block validator. However, it is important to acknowledge that employing the Randao library or relying on the block.prevrandao value introduces certain vulnerabilities similar to those associated with using the previous block hash. One such drawback is that a block validator retains the ability to cancel the validation of a block if the prevrandao value is not advantageous to their interests (miner incentive problem) [15].

Again like with the block hash, one can choose to select the Randao value of a future block to exclude the miner incentive problem. To retrieve the value of future block ‘n’, the developer has to wait until block ‘n’ has passed and retrieve it then. However, retrieving the Randao value of a future block presents a challenge.This is because it is possible to implement prevrandao(block ‘n’) once block ‘n’ has already passed. The request must be performed in block ‘n’. A potential solution exist out of the selection of the prevrandao value from block ‘n’ or any subsequent block. However, validators can then influence the random number even more by not including the transaction in their block [15].

However, it is extremely unlikely that a block validator would simultaneously purchase an NFT from the collection at that moment. Furthermore, the validator is not yet aware of the collection’s rare traits. This requires the knowledge of the prevrandao numbers of the blocks of all mint transactions. As a result, choosing the prevrandao value of the current block to generate a random number for NFT collections is an appropriate choice.

6.6 Randomness in existing on-chain collections

6.6.1 OnChainMonkeys

The OnChainMonkeys, which have 9500 items and a trade volume of already 17 253 ether as of April 2, are the most well-known on-chain NFT collection. Nevertheless, there is no on-chain randomization method present in this collection. In fact, it is feasible to determine in advance which ids correspond to particular traits. Listing 6.1 shows the technique used to translate a `tokenId` into several `traitIds`. First, a private string and the `tokenId` are concatenated. On the blockchain, this secret string is visible and the `tokenId` is also well-known. The hash is then extracted from this text and transformed into an `uint256`. The appropriate `traitId` is derived by taking the modulo. Hence, unlike other off-chain collections, this collection lacks a pre-reveal phase and is fairly predictable [14].

Listing 6.1: traitId generator OnChainMonkeys

```

string private ra1 = "A";

backgroundId = uint8(random(string(abi.encodePacked(
                  ra1,tokenId.toString()))) % 8);

function random(string memory input) internal pure returns (
    uint256) {
    return uint256(keccak256(abi.encodePacked(input)));
  }
```

6.6.2 OnChainBirds

The OnChainBirds were launched in September of 2022 and have 10,000 items, with 767 ether worth of trading volume on April 3. Unlike the OnChainMonkeys, this collection did manage to randomly generate the `traitIds` on the blockchain. Listing 6.2 provides an illustration of the random number generator. For every `tokenId`, a random number is produced. All of the `traitIds` are then calculated from this random number. The `tokenId`, block difficulty, block timestamp, and `msg.sender` are the parameters used to produce the random number. This final field specifies the address of the transaction's performer, in this instance the buyer. Both the `tokenId` and the buyer's address are visible for the buyer. However, In light of the first two variables, it may be concluded that the customer is unsure of the NFT they will purchase [12].

A crucial detail is that the first mint transaction occurred on September 16, 2022, one day after the Etherreum merge. The block difficulty did not instantly yield a value of zero at this point. It was decided to switch the underlying opcode to the new prevrandao PoS field [12].

LISTING 6.2: random number generator OnChainBirds

```
uint256 randinput =
                uint256(
                    keccak256(
                        abi.encodePacked(
                            block.timestamp,
                            block.difficulty,
                            tokenId,
                            msg.sender)));
```

6.6.3 OnChainDonalds

The randomization of attributes was handled even differently by the OnChainDonalds. They opted for the reveal phase option rather than a reveal at mint. This allowed them to randomize all the traits with one particular random seed at the same time. With this approach, the developer determines the random seed off-chain before passing it online to the smart contract. Listing 6.3 describes the approaches to take in order to achieve this. The benefit of using this technique is that the seed may be checked first off-chain to ensure that duplicates are not introduced. However, this technique again introduces trust into the developer. After all, the developer is not allowed to share the seed with other people beforehand [13].

LISTING 6.3: random number generator OnChainDonald

```
function reveal(uint256 _seed) external onlyOwner {
        require(state == STATE_MINTING);
        require(_seed != 0);

        state = STATE_MINTING_REVEALED;
        seed = _seed + 0x201620202024c0fefe;
    }

function random(uint256 _seed, string memory input)
        private
        pure
        returns (uint256)
    {
        return _seed + uint256(keccak256(abi.encodePacked(input)))
           ;
    }
```

6.7 Conclusion

It can be concluded that despite the deterministic nature of the blockchain, there are still options to generate random numbers on-chain. To choose which solution is the most appropriate, the context must be taken into account. For instance, for a lottery security implications are far more crucial than those for an NFT collection. Therefore, this security issue requires more consideration. In cases like these, a commit-reveal principle with homomorphic encryption is the best option. However, this solution is too cost-prohibitive and requires too much interaction for an NFT collection. It is therefore better to choose a commit-reveal library like RANDAO. Due to the removal of the homomorphic calculation, this is a cost-effective alternative and furthermore, interaction is not necessary. The RANDAO library can be used most inexpensively using the prevrandao value of the current block. Although choosing an oracle is a safe option, choosing a reveal at mint will be too costly. In this case, a random number is required for each NFT. Finally, it should be mentioned that block information, except the prevrandao field, should not be used because it can be easily manipulated.

Chapter 7

Sampling of NFTs in a Collection

Sampling is a method frequently used to choose a representative group from a population. In this case, the sample must reflect the entire population. There are several known techniques and algorithms to accomplish this. These methods can also be used for picking the specific NFTs used in a collection from all the possible combinations created by the trait-combinations. For example, with the bunny collection 5000 items will be selected from 80 640 000 possible items. This selected sample must comply with the prescribed rarity rules. This section compares different sampling techniques in order to exclude insider trading and decrease deployment cost.

7.1 Sampling method Loot

The sampling algorithm used in the Loot collection was also applied to the OnChain-Bunnies and is illustrated for calculating the `whiskerId` in Listing 7.1. Each trait is associated with an array stored as storage data, containing weights assigned to each `traitId`. The selection of traits starts in `randomOne(uint256 tokenId)` method, where a random number is generated as discussed in Section 6. The sum of weights, in this case 100 for whiskers, is taken and the random number is modulo-divided by this sum to determine the effective weight of the `traitId`. The `usew(uint8[] memory w, uint256 i)` then calculates which category of `traitId` this weight falls into [14, 11].

As mentioned earlier in Section 6, the random number generated in the Loot and OnChainMonkey contract is not truly random but predictable. The reason for this is that the developer can calculate in advance the combination of `traitIds` and ensure the absence of duplicates. These duplicates arise due to the weights. For example, all calculated weights between 0-50 for whiskers will result in `whiskersId` 1. If this happens by chance for all the weights duplicates are introduced. This probability is very high because of the very scattered weights. For example, as shown in Listing 7.1, the probability of `whiskerId` zero is 50%. To address this issue, the implementation includes adjustable parameters to mitigate duplicate occurrences.

For instance, in the `randomOne(uint256 tokenId)` method, the `tokenId` is subtracted from a fixed value, and a private seed (e.g., the letter ‘H’ for whiskers) is incorporated into the random computation. Finally, by adjusting the parameters, a collection could be generated with a single duplicate. The specific duplicate with `tokenId` 7403 was finally excluded by increasing the`hatId` by one [14].

The sampling algorithm of the Loot collection applied to the OnChainBirds, with the application of previously mentioned optimizations, results in a deployment cost of 7 485 689 gas. This cost is partially attributed to the computation of the mapping from `tokenId` to `traitIds`, as the while loop within the `usew` method needs to be executed for each `traitId`. Moreover, insider trading is possible since the mapping from `tokenId` to `traitIds` can be calculated off-chain. However, this does provide a certainty of excluding duplicates.

Listing 7.1: Sampling algorithm Loot applied to OCB for selcting whiskerId

```
uint8[] private whiskers_weight = [50,25,20,5];

function usew(uint8[] memory w,uint256 i) internal pure returns (
    uint8) {
        uint8 ind=0;
        uint256 j=uint256(w[0]);
        while (j<=i) {
        ind++;
        j+=uint256(w[ind]);
        }
        return ind;
    }

function randomOne(uint256 tokenId) internal view returns (Bunnie
    memory) {
        tokenId=10000-tokenId;
        Bunnie memory bunnie;
        ...
        bunnie.whiskers = usew(whiskers_weight,random(string(abi.
            encodePacked('H', tokenId)))%100);
        ...
      if (tokenId==7403) {
        bunnie.hat++;
      }
     return bunnie;
```

7.2 Insider trading

One of the problems solved in this section is the problem of insider trading in NFT collections. This problem occurs due to the fact that the sampling of the combination of traits is predictable. This predictability can be attributed to two methods. Firstly, in off-chain collections, where the images are stored with a third party, the sampling occurs off-chain in advance. The developer assembles the items before deployment and stores them at a remote server or via a distributed file system. Secondly, in the case of the Loot collection, the code becomes publicly available upon deployment, allowing anyone to calculate the exact images that will be generated. In both scenarios, it is clear which `tokenId` will result in which `traitIds`. Secondly, in the case of the Loot collection, the code becomes publicly available upon deployment, allowing anyone to calculate the exact images that will be generated. Anyone who is aware of the method can perform the calculation and act in a manner that benefits them.

In both cases, it is evident that which `tokenId` results in which image. However, even if a person possesses this information, they still need to be able to purchase the correct image. How this works and how easy this is depends on the method of reveal. To illustrate these methods, let's consider an example where a person is aware that tokenId 1723 corresponds to a rare and valuable NFT.

A first method that can be implemented is a reveal at mint. In this case, the image is revealed immediately upon mint. The `tokenId` serves as a counter in the minting process, ensuring that the first buyer owns `tokenId` 1, and subsequent buyers are assigned incremental `tokenIds`. So in this case, the person with the insider information has to wait until he is the 1723 buyer and mints at that specific time. This is not always easy due to the fact that there may be other buyers at the same time. A second option for reveal works with a reveal phase. During mint, the `tokenId` is calculated in the same manner as before, but the image and traits are not immediately visible. Nevertheless, the image is already for sale before the reveal. If a person with insider info accidentally minted `tokenId` 1724 because there was another buyer at the same time, the person can still offer a higher price for `tokenId` 1723. The unsuspecting buyer who sees this offer will likely accept it, unaware of the item's value.

To counter the problem of insider trading, an algorithm can be constructed to sample the items from all possible `sampleIds` in a random manner at the time of the mint or at reveal, depending on the method chosen. It was chosen to work with a reveal at mint since it is trivial to convert it to a reveal phase. In this case the sampling algorithm is performed at a different moment in the process. Making the sampling random ensures that neither the developer nor the buyer can calculate which `tokenId` results in which `traitIds`. However, to perform this sampling algorithm, an assembling algorithm like that of Loot needs to be employed. Without the assembling algorithm, all possible combinations would have to be stored, which is an unnecessary storage cost. It is possible to store the traits off-chain and use the Loot algorithm on-chain. However, this is not recommended due to retrieval delays.

Another option to solve the insider trading problem is to make it unpredictable which `tokenId` results in which image. This can be obtained by applying a shuffle method to the `tokenIds`. So in this case, the first person to mint will not necessarily get `tokenId` 1. However, this is a costly way of implementation since it must be checked that an item is not selected twice. Therefore, this thesis does not discuss this method further.

7.3 Mapping

Before looking into various sampling methods, it is crucial to determine the specific population from which the sample will be drawn. In the context of the Loot smart contract, sampling is performed on each `traitId`. In this case, sampling is performed nine times for each item which is a costly task. Furthermore, if random numbers are chosen for sampling, it becomes necessary to store a mapping that maps each `tokenId` to the combination of `traitIds`. This ensures that when an image is subsequently requested, the appropriate traits can be assembled correctly.

An alternative solution is to sample from all possible combinations, which amounts to 80 640 000 for the OnChainBunnies. In this case, the sampling must be calculated only once. To facilitate this approach, it is essential to maintain a mapping in the storage data that links each `tokenId` with a `sampleId`. Here, the `sampleId` represents the sampled value.

The choice to sample from `sampleIds` instead of `traitIds` offers two advantages. First, it is a more cost-effective solution because of a smaller mapping is stored and the number of sampling computations is divided by the number of traits. Secondly, sampling `sampleIds` provides the flexibility to exclude duplicates if a one-on-one mapping approach is employed, and the chosen sampling method ensures that the same `sampleId` is not selected twice. This thesis present two distinct options for such a one-one-one mapping.

7.3.1 Modulo Mapping

A unique mapping between the `sampleId` and distinct traitIds is required. Moreover, this mapping must remain unpredictable to prevent any potential exploitation. As a first type of mapping, a modulo mapping is presented. An example is illustrated in Figure 7.1. In the example the `purseId` is calulated as `sampleId` % 9, while the `tieId` is calulated as `sampleId` % 2 764 800 (8*6*5*4*4*30*24). However this mapping is predictable. This is because a buyer will know that `sampleIds` less than 2 764 800, will yield in an item with `tieId` equal to zero. If the buyer knows the `sampleIds` ascend, which is the case for some algorithms later discussed, and the buyer wants a tie with id equal to 7, he just waits long enough until the `sampleId` is higher than 2 764 800 * 7. To mitigate this issue, a mapping using bitfield encoding will be adopted.

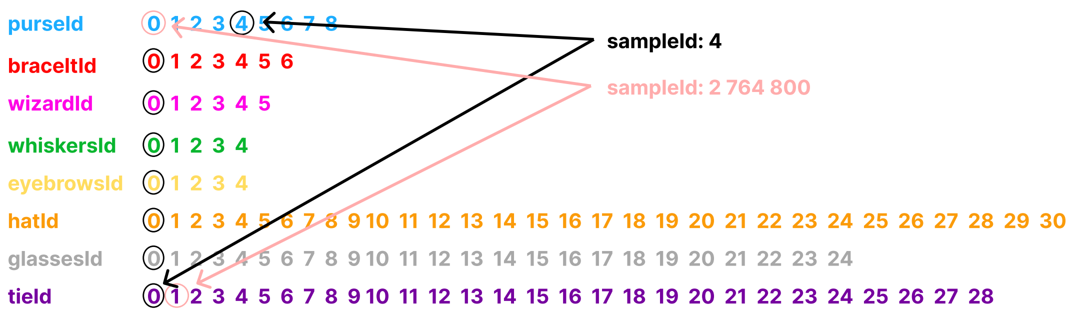

purseId 0 1 2 3 4 5 6 7 8
braceltId 0 1 2 3 4 5 6
wizardId 0 1 2 3 4 5
whiskersId 0 1 2 3 4
eyebrowsId 0 1 2 3 4
hatId 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
glassesId 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
tieId 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
sampleId: 4
sampleId: 2 764 800

Figure 7.1: Modulo mapping

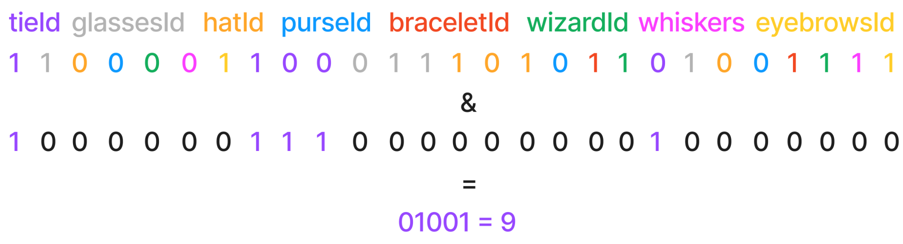


Figure 7.2: BitfieldEncoding of tieId

7.3.2 Bitfield encoding Mapping

In bitfield encoding traits are extracted from a single `uint256 id`, with each trait represented by a specific number of bits. Figure 7.2 is an illustration of an example. The `tieId`, in this case, exists of 5 bits, which correspond to $2^5(32)$ distinct values. In the example the bits that assemble the `tieId` are located at index 8, 19, 20, 21 and 28 respectively. Consequently, the `tieId` can differ for each consecutive `sampleId`, ensuring uniqueness. Moreover, this method guarantees low-cost storage. Instead of seven separate `traitIds`, only one id has to be stored for each NFT.

7.4 Sampling with replacement

Among the various sampling methods, two types are provided. The first is referred to as sampling with replacement. This means that after an NFT is chosen from the population, sampling will continue with a population in which the chosen NFT is put back in the sampling pool. In other words, the sample can contain duplicates. Sampling without replacement suggests the opposite. After selecting the NFT, the NFT is not put back into the sampling pool. This means that an item can not be selected twice, therefor duplicates can not occur. First, some sampling techniques with replacement are contrasted.

Algorithm 1 Inverse transform method with exponential distribution

```
1: procedure InverseTranfsormExp
2:     $mean \leftarrow 1$
3:     $u \leftarrow u \sim (0, 1)$
4:     $id \leftarrow floor(-mean * log(1 - u))$
5:     while $id \geq= 80640000$ do
6:         $mean \leftarrow 1$
7:         $u \leftarrow u \sim 1$
8:         $id \leftarrow floor(-mean * log(1 - u))$
9:     end while
10:    return $id$
11: end procedure
```

7.4.1 Inverse transform method

The inverse transform method is an algorithm that samples from a distribution function P(X). The first step in the algorithm exists out of generating the cumulative distribution F(X). The cumulative distribution function for a random variable X can be described by [56]:

$$F_X(x) = P(X \leq x) = \int_0^x P(x')\, dx', x \in \mathbb{R}$$

In step two of the algorithm the inverse of the cumulative distribution is calculated together with a random number uniformly distributed on [0,1]. Then the sampling X is calculated by the following equation [56]:

$$X = F_X^{-1}(U)$$

A proof of the algorithm is provided by Karl Sigman [53] and can be found in A. The implementation of such a sampling with a exponential distribution function can be found in Algorithm 1. This algorithm will be executed every time a new mint transaction is performed.

The distribution function can be chosen from a variety of alternatives. Calculating the matching CDF for each distribution function is not always simple. As a result, a number of straightforward distribution functions are given and contrasted [53].

Exponential distribution

The exponential probability density function is denoted by the following formula [53]:

$$F(x) = \lambda e^{-\lambda x}, x \geq 0, \lambda > 0$$

Setting this function equal to y and solving this for x in terms of $y \in (0,1)$, gives the following formula for the CDF [53]:

$$x = -\frac{1}{\lambda} ln(1-y)$$

Then, for each sampling cycle, a uniform random integer between 0 and 1 is generated. The `sampleId` is then determined by applying $X = -\frac{1}{\lambda} ln(U)$. Here $ln(1-U)$ is replaced by $ln(U)$ since it holds that $1 - U\ unif(0,1)$ and $U\ unif(0,1)$ [53].

The method was implemented in python to test in how many cases duplicates occurred. In each test, the entire collection was sampled and tested for the occurrence of duplicates. Each experiment consisted of running the test 1000 times. First, the inverse sampling method was employed using an exponential distribution function, with the mean parameter set to one, for each trait. This showed that duplicates occurred in 6.7 % of the cases. Next, the experiment was performed on the `sampleIds` with the bitfield encoding mapping. This showed that there is 21.9 % chance of duplicates. This is due to the fact that exponential division is not performed on each trait individually but mainly on the `tieId` since it represents the largest bit. As a result, the probability of a `tieId` greater than 15 (`b01111` as a bitfield representation) will be very small which significantly increases the probability of duplicates.

An exponential distribution is a good solution to introduce rare items in a collection and also a cost-effective choice since only one calculation has been made. However, a logarithmic calculation in Solidity is still costly.

Discrete distribution

Assume that the distribution follows a probability mass function $p_i = P(X = x_i)$. In this case the CDF $X = F^{-1}(U)$ is described by [33]:

$$X = 0, if U \leq p(0)$$

,

$$X = x_k, if \sum_{j=1}^{k-1} p_j \leq U < \sum_{j=1}^{k} p_j$$

The algorithm then exists out of two steps. First, to create a random uniform number between zero and one and second, finding k such that the above equation yields [33]. However, this second step requires a search, which is an expensive operation. Furthermore, the probability mass function is an array of the same length as the number of items the sample is taken from. Storing this array is an expensive operation as well.

This method is similar to the one implemented in the Loot and OnChainMonkeys collection, only in this case a truly random number was selected. The `usew(uint8[]`

`memory w, uint256 i)` in Listing 7.1 shows the implementation. Here w represents the probability mass function and i the random number.

Again, the method was tested in python. Applying the sampling technique to the different traits results in the fact that in 99.5% of the cases the collection contains duplicates. However, it is essential to note that this outcome heavily relies on the specific mass function selected and should not be regarded as a universally applicable guideline. The sampling technique was not applied to the full sample size since the mass function consisted of an array of 80 640 000 integers, which is not possible to store.

There can be stated that a discrete distribution can definitely introduce rare traits because the probability of every trait can be specified. However, it is an expensive choice due to the storage of the probability mass array and the search operation. Finally, the chance op duplicates is extremely high.

Discussion of inverse transform method

The exponential distribution function is the most affordable alternative for distribution function sin the inverse transform method and introduces the option for rare traits. However, there is less control over the rarities of the different traits in contrast to a discrete distribution.

As a result, the inverse transform method with an exponential distribution is a viable alternative for sampling an NFT collection out of a large collection of NFTs. In fact, it is completely unpredictable. However, as it is a sampling approach without replacement, it could introduce duplicates.

7.4.2 Acceptance-reject method

The acceptance-reject approach is the focus of the second sampling algorithm with replacement. This method examines sampling according to a distribution function, where calculating the CDF is not always straightforward. The algorithm is represented in Algorithm 2. The basic idea behind acceptance-reject sampling is illustrated in 7.3. Candidate values are generated from a proposal distribution that is relatively easy to sample from, in the example a normal distribution, and the each candidate gets accepted or rejected based on whether it falls within the desired target distribution. The acceptance decision is made by comparing the ratio of the target distribution's density function to the proposal distribution's density function with a uniform random variable. If the candidate is below the acceptance ratio, shown in green in Figure 7.3 [55], it becomes part of the random sample. If the sample falls in the red rejection ratio, it is discarded, and the process is repeated until a sufficient number of samples are obtained [29]. A proof of the algorithm is provided by Bernard Flury [38] and can be found in Appendix A.

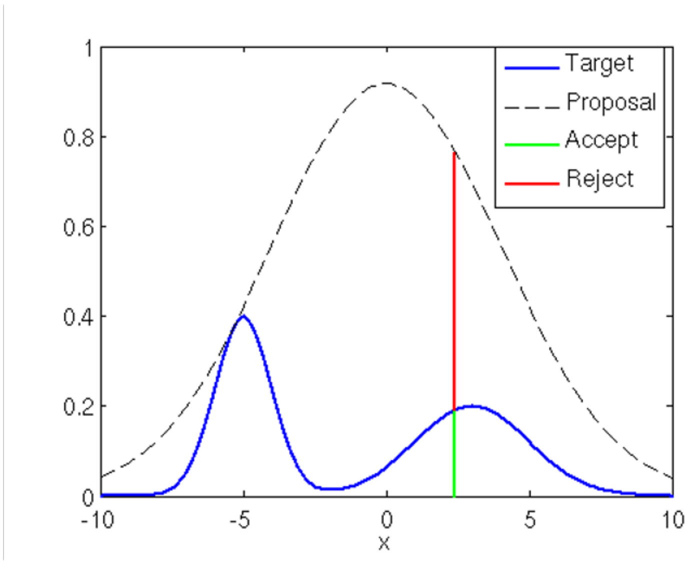


FIGURE 7.3: Acceptance-reject sampling

However, the acceptance-rejection method is an expensive method in terms of gas cost. Both the evaluation of the target and proposal function must be calculated. Furthermore, two random numbers must be generated per sampling which is also more expensive than the inverse transform method.

7.5 Sampling without replacement

7.5.1 Knuth

Knuth's algorithm is the first algorithm of sampling techniques without replacement to be examined. The technique creates a uniform random number between zero and one at each sampling round. Afterwards it will sample ids in an ascending order, uniformly distributed over the population size. The sample size and population size represent the input parameters. Since the `sampleIds` increase each time a new NFT

Algorithm 2 Acceptance-reject sampling

```
1: procedure InverseTranfsormExp
2:     $id \leftarrow u \sim (0, 80639999)$
3:     $u \leftarrow u \sim (0, g(id))$
4:     while $u > f(id)$ do
5:         $id \leftarrow u \sim (0, 80639999)$
6:         $u \leftarrow u \sim (0, g(id))$
7:     end while
8:     return $id$
9: end procedure
```

is minted, there is an assurance that no duplicates will occur [45]. A proof of the algorithm is provided by Knuth [45] and can be found in Appendix A.

The algorithm is used during mint and shown in Listing 7.2. It loops over all possible combinations. The Knuth parameter represents which `sampleId` is checked at that moment in the loop. For every `sampleId` the following equation is checked:

$$(populationSize - Knuth\_parameter) * u >= sampleSisze - nrItemsSelected$$

The sample size is represented by the `MAX_PURCHASE` parameter and the number of items that has been selected so far by the supply parameter. The parameter u is a uniform random number between zero and one. Because of the fact that Solidity can only store integers there has been chosen to generate a random number between zero and 99999 and multiply the right hand side of the equation with 99999.

However, it is important to acknowledge a limitation regarding the ascending order of `sampleIds` in combination with the bitfield encoding mapping. As explained earlier, the usage of bitfield encoding makes it unpredictable which `tokenId` will result in which `traitIds`. However, it can be predicted in which range some `traitIds` will fall. For instance, if the sampleId is less than 2^{17}, it ensures that `tieId` values higher than 15 will not occur. Here, 2^{17} represents the numerical value when the first bit of `tieId` is set. This is because 18 bits have been used to represent the traits and the first bit of `tieId` corresponds to the 18th bit from the `sampleId`. The number 15 corresponds to the bit representation `b01111`, representing the highest `tieId` that can be assumed without the first bit being set. For further research, a more unpredictable one-to-one mapping can be sought.

The fact that the selection is random and does not introduce any duplicates gives the Knuth algorithm a substantial advantage over the Loot algorithm. However, this method requires looping over all possible combinations. If this is implemented in Solidity, when the mint method is called, the smart contract will run out of gas on which the transaction reverts. This makes it impossible to use the algorithm on the blockchain.

Listing 7.2: Sampling algorithm Knuth

```
uint256 public Knuth_parameter = 0;
mapping(uint256 => uint256) private IdToSampleId;

function mint(uint256 numberOfTokens) external {
        uint256 populationSize = 268435456;
        uint256 supply = totalSupply();
        while (numberOfTokens != 0 ) {
            if ((populationSize - Knuth_parameter)*createRandom(
                supply) >= (MAX_PURCHASE-supply) *99999){
                 Knuth_parameter++;
            }

            else if (checkId(Knuth_parameter)){
                _mint(msg.sender, supply + 1);
                IdToSampleId[supply + 1] = Knuth_parameter;
                numberOfTokens--;
                Knuth_parameter++;
            }
            else{
                Knuth_parameter++;
            }

        }
```

7.5.2 Variant Knuth

The above Knuth algorithm cannot be used because of the looping over the population size. However, the algorithm can be modified to allow looping in larger increments. This can be accomplished by storing the previous minted `sampleId`, illustrated in Listing 7.3 as the `last_selected` parameter. A random number is then generated between 1 and the sample size divided by the population size. The sum of the random number and previous minted `sampleId` then corresponds to the new `sampleId`. This way the while loop loops over the sample size instead of the population size. However, there is again the introduction of a uniform distribution so that rare traits do not occur. Furthermore, the usage of bitfield encoding in this case leads to a semi-predictable outcome, as explained in the Knuth algorithm above. The implementation of this algorithm in Solidity results in a deployment cost of 6 692 382 gas, which is a reduction of 10.65% in contrast to the optimizations from the previous chapter.

LISTING 7.3: Sampling algorithm variant Knuth

```
uint256 public last_selected = 0;
mapping(uint256 => uint256) private IdToSampleId;

function mint(uint256 numberOfTokens) external {
        uint256 supply = totalSupply();
        while (numberOfTokens != 0 ) {
            uint256 sampleId = last_selected +
                createRandomBetween1And25687(supply+1);
            if (checkId(sampleId)){
                _mint(msg.sender, supply + 1);
                supply++;
                IdToSampleId[supply + 1] = sampleId;
                numberOfTokens--;
                last_selected = sampleId;
            }
            else{
                last_selected = sampleId;
                }

            }
        }
```

Proof

The proof of the algorithm exists out of two parts and is based on the proof provided by Knuth [45] in the previous section. As a first part there can be verified that there occur no duplicates in the collection when using the sampling algorithm. This can be simply justified by the fact that the `sampleIds` are picked in ascending order and there is one-to-one mapping from `sampleId` to `traitIds`. As a second part there can be proven that at most N records are input. This means that a `sampleId` higher than the sample size will never be picked. This can be confirmed by the fact that the maximum difference between two `sampleIds` will be the sample size divided by the population size. If the maximum difference is always chosen at random, the maximum `sampleId` will be equal to sample size. There can not be proven that the distribution is uniform because this, in contracditcion with Knuth's algorithm, is not the case. Small `sampleIds` have a higher chance to be picked then higher `sampleIds`.

7.6 Conclusion

To counter insider trading, an on-chain sampling algorithm that uses a random number is necessary. Initially, a mapping was established to link each `sampleId` with a distinct combination of `traitIds`. In this regard, bitfield encoding emerged as the most unpredictable and cost-effective solution. Next, different sampling methods were compared. A sampling method without replacement is the best option because of the exclusion of duplicates. Knuth's sampling method underwent modifications to ensure that the mint method did not encounter gas depletion issues. The modified algorithm combined with bitfield encoding as a mapping provided a 10.65% reduction in deployment cost.

Chapter 8

Conclusion and further research

In the initial phase of this thesis, a low-cost way of storing an NFT collection with the absence of traits was investigated. In the scope of experimental research, a dataset consisting of 100 images was utilized, which was generously provided by an external artist. The images exhibited variations in size, ranging from a minimum of 0.5 kilobytes to a maximum of 100 kilobytes. First, a method was devised to inject the SVG images as bytecodes and storing these images in a mapping. The smart contract was optimized using the Solidity Optimizer with a parameter of 200 runs. Extensive image optimizations were performed, including rounding or rewriting paths, merging paths, and removing style elements and backgrounds. Notably, the rewriting of paths yielded the most significant reduction in price. However, this was a noisy compression technique and can not be used in every case. Collectively, these optimizations led to a remarkable cost reduction by a factor of 6.13 or an average price reduction of 83.68 %. In the future, to optimize this reduction even more, an additional storage location, namely that of events, could be considered. Moreover, it would be worthwhile to explore a wider range of assembly optimizers and even consider developing a custom optimizer implementation.

Moving forward, the thesis delved into NFT collections incorporating traits. Here, the Loot algorithm was applied to a custom-developed collection. The Loot algorithm stores the traits in storage data and generates the images on-chain. The application of the algorithm resulted in an average reduction of 99.88% in deployment cost. To further enhance cost efficiency, several measures were implemented, including transferring strings from storage to memory data, utilizing the Solidity optimizer, and implementing a sampling algorithm with a bitfield encoding mapping. These combined efforts yielded a further reduction in price to an impressive 99.95 %. To improve this even more, the same optimizations mentioned in the previous paragraph can be considered. Additionally, it is important to note that pixel art NFT collections have not been taken into account thus far. Transforming the images to a lower resolution allows the traits themselves to be generated on-chain using the SVG path element.

Finally, a solution was offered to the insider trading problem that regularly occurs with NFT collections. This insider trading is caused by the fact that the developer owns the information which `tokenId` results in a rare NFT. To counter this, a sampling algorithm without replacement was proposed. This algorithm is based on Knuth's algorithm. Furthermore, the usage of a bitfield encoding mapping was Incorporated. This was implemented so that sampling without replacement would be less predictable. Currently, the buyer can not predict which specific `traitId`s he will get but he can predict in what range they will fall. More research on an unpredictable one-to-one mapping can still be performed. Finally, the algorithm uses a random number. Various options were considered to produce this random number as cheaply and unpredictable as possible on the blockchain. Ultimately, it was determined that utilizing the prevrandao field in combination with the block hash, `tokenId`, and buyer's address provided the most suitable solution. This method used can still be affected by the block validators. More in depth research to solve this problem can be performed.

Appendices

Appendix A

Proofs of Sampling Algorithms

These proofs are provided by Karl Sigman [53], Bernard Flury [38] and Knuth [45] for proving the inverse transform method, acceptance-rejection method, and Kuth's algorithm respectively.

A.1 Inverse transform method

Lemma: Let $F(x), x \in \mathbb{R}$, denote any cumulative distribution function (cdf) (continuous or not). Let $F^{-1}(y), y \in [0,1]$, denote the inverse function defined in

$$F^{-1}(y) = min\{x : F(x) \geq y\}, y \in [0,1]$$

. Define $X = F^{-1}(U)$, where U has the continuous uniform distribution over the interval (0,1). Then X is distributed as F, that is $P(X \leq x) = F(x)$, $x \in \mathbb{R}$.

Proof: We must show that $P(F^{-1}(U) \leq x) = F(x), x \in \mathbb{R}$. First suppose that F is continuous. Then we will show that (equality of events) $\{F^{-1}(U) \leq x\} = \{U \leq F(x)\}$, so that by taking probabilities (and letting $a = F(x)$ in $P(U \leq a) = a$) yields the result: $P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x)$.
To this end: $F(F^{-1}(y)) = y$ and so (by monotonicity of F) if $F^{-1}(U) \leq x$, then $U = F(F^{-1}(U)) \leq F(x)$, or $U \leq F(x)$. Similarly $F^{-1}(F(x)) = x$ and so if $U \leq F(x)$, then $F^-1(U) \leq x$. We conclude equality of the two events as was to be shown. In the general (continuous or not) case, it is easily shown that

$$\{U < F(x)\} \subseteq \{F^{-1}(U) \leq x\} \subseteq \{U \leq F(x)\},$$

which yields the same result after taking probabilities (since $P(U = F(x)) = 0$ since U is a continuous rv.)

A.2 Acceptance-reject method

We shall state three lemmas, each of them rather trivial, but together they provide an elegant proof that the acceptance-rejection method works. The only prerequisite for understanding the proof is the notion of conditional distribution.

Lemma 1. Let X denote a p-variate random variable with density g(x), let c > 0 denote a real constant, and let Y denote a random variable such that the conditional distribution of Y, given X=x, is uniform in the interval (0,cg(x)). Then the joint distribution of (p+1)-dimensional random variate $\binom{X}{Y}$ is uniform in the set

$$\mathscr{A} = \{\begin{pmatrix} x \\ y \end{pmatrix}; x \in \mathbb{R}^p, 0 < y < cg(x)\}.$$

Proof. Let $h(x, y)$ denote the joint density of X and Y. Then

$$h(x,y) = g(x) \cdot \frac{1}{cg(x)} = \frac{1}{c} \text{ if } \begin{pmatrix} x \\ y \end{pmatrix} \in \mathscr{A}$$

$$= 0, \text{ else .}$$

In Lemma 2, $V(\cdot)$ will denote the m-dimensional volume of a set.
Lemma 2. Suppose the m-dimensional random variable Z has a uniform distribution in $\mathscr{A} \subset R^m$, where $0 < V(\mathscr{A}) < \infty$. Let $\mathscr{B} \subseteq \mathscr{A}, V(\mathscr{B}) > 0$. The the conditional distribution of Z, given $Z \in \mathscr{B}$, is uniform in $\mathscr{B}$.
Proof. The proof is obvious.

Lemma 3. Suppose (p+1)-dimensional random variable $\binom{X}{Y}$, where X has dimension p, follows a uniform distribution in the set $\mathscr{B} = \{\binom{x}{y} : x \in \mathbb{R}^p, 0 < y < f(x)\}$, where f is the density function of a p-variate random variable. Then the marginal distribution of X has density f.
Proof. The density of X at the point $x \in \mathbb{R}^p$ is $\int_0^{f(x)} dy = f(x)$. Putting the lemmas together, with $m = p + 1$ in Lemma 2, proves the acceptance-rejection method.

A.3 Algorithm Knuth

We can verify that
A) At most N records are input (we never run off the end of the file before choosing n items).
B) The sample is completely unbiased; in particular, the probability that any given element is selected, e.g., the last element of the file, is n/N. Statement (b) is true in spite of the fact that we are not selecting the (t+1)st item with probability n/N, we select it with the probability in

$$\frac{\binom{N-t-1}{n-m-1}}{\binom{N-t}{n-m}} = \frac{n-m}{N-t}$$

We will usually not have to pass over all N records. In facts, since (b) above says that the last record is selected with probability n/N, we will terminate the algorithm before considering the last record exactly $(1 - n/N)$ of the time.

Bibliography

[1] The ethereum proof-of-stake merge. URL: https://ethmerge.com/, 2021. Last accessed 20 november 2022.

[2] Arcana: The web3 privacy stack. URL: https://arcana.network, 2022. Last accessed 12 december 2022.

[3] Arweave: Store data, permanently. URL: https://www.arweave.org, 2022. Last accessed 12 december 2022.

[4] Filecoin is a decentralized storage network designed to store humanity's most important information. URL: https://filecoin.io, 2022. Last accessed 12 december 2022.

[5] Ipfs powers the distributed web. URL: https://ipfs.tech/, 2022. Last accessed 12 december 2022.

[6] The optimizer. URL: https://docs.soliditylang.org/en/v0.8.17/internals/optimizer.html, 2022. Last accessed 13 december 2022.

[7] Pinata: Your home for nft media. URL: https://www.pinata.cloud/, 2022. Last accessed 12 december 2022.

[8] Storj: Fast, secure cloud storage at a fraction of the cost. URL: https://www.storj.io, 2022. Last accessed 12 december 2022.

[9] Chainlink docs. URL: https://docs.chain.link/, 2023. Last accessed 23 february 2023.

[10] Gas. URL: https://ethereum.org/en/developers/docs/gas, 2023. Last accessed 23 february 2023.

[11] Loot smart contract. URL: https://etherscan.io/address/0xff9c1b15b16263c61d017ee9f65c50e4ae0113d7#code, 2023. Last accessed 23 february 2023.

[12] Onchainbird smart contract. URL: https://etherscan.io/address/0xbe82b9533ddf0acaddcaa6af38830ff4b919482c#code, 2023. Last accessed 23 february 2023.

[13] Onchaindonald smart contract. URL: https://etherscan.io/address/0xbd46b75289661765873c8c6d5051f20a6bfb5335#code, 2023. Last accessed 4 may 2023.

[14] Onchainmonkey smart contract. URL: hhttps://etherscan.io/address/0x960b7a6bcd451c9968473f7bbfd9be826efd549a#code, 2023. Last accessed 23 february 2023.

[15] Solidity deep dive: New opcode 'prevrandao'. URL: https://soliditydeveloper.com/prevrandao, 2023.

[16] Svgo v3.0.0. URL: https://jakearchibald.github.io/svgomg/, 2023. Last accessed 23 february 2023.

[17] Types. URL: https://docs.soliditylang.org/en/v0.5.11/types.html, 2023. Last accessed 23 february 2023.

[18] C. A.B. Erc-4883. https://eips.ethereum.org/EIPS/eip-4883, 2022.

[19] P. Allemann, C. Cachin, L. Zanolini, and I. A. Sesar. Randomness and games on ethereum. 2021.

[20] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[21] T. Baldwin and M. Sander. *De kracht van blockchain: systematiek van de toekkomst*. Mijnmangementboek.nl, 2021.

[22] S. Barrington and N. Merrill. The fungibility of non-fungible tokens: A quantitative analysis of erc-721 metadata. *arXiv preprint arXiv:2209.14517*, 2022.

[23] I. Bashir. *Mastering Blockchain: A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more*. Packt Publishing Ltd, 2020.

[24] Beeple. Everydays: The first 5000 days. URL: https://opensea.io/assets/ethereum/0x2a46f2ffd99e19a89476e2f62270e0a35bbf0756/40913, 2021. Last accessed 20 november 2022.

[25] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs*, 1, 2021.

[26] A. BTCManager. The blockchain lottery smartbillions was hacked for 120,000 dollar. 2017.

[27] C352B5. Cryptopunks collection. URL: https://opensea.io/collection/cryptopunks, 2017. Last accessed 20 november 2022.

[28] S. Casale-Brunet, P. Ribeca, P. Doyle, and M. Mattavelli. Networks of ethereum non-fungible tokens: a graph-based analysis of the erc-721 ecosystem. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 188–195. IEEE, 2021.

[29] G. Casella, C. P. Robert, and M. T. Wells. Generalized accept-reject sampling schemes. *Lecture Notes-Monograph Series*, pages 342–347, 2004.

[30] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani. Probabilistic smart contracts: Secure randomness on the blockchain. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 403–412. IEEE, 2019.

[31] J. Chittoda. *Mastering Blockchain Programming with Solidity: Write production-ready smart contracts for Ethereum blockchain with Solidity.* Packt Publishing Ltd, 2019.

[32] E. Daniel and F. Tschorsch. Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks. *IEEE Communications Surveys & Tutorials*, 24(1):31–52, 2022.

[33] L. Devroye and L. Devroye. Discrete random variates. *Non-Uniform Random Variate Generation*, pages 83–117, 1986.

[34] A. Di Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora. Profiling gas consumption in solidity smart contracts. *Journal of Systems and Software*, 186:111193, 2022.

[35] S. Dickens. Nft collection raccoon secret society kills entire project to prove a point. URL: https://finance.yahoo.com/news/nft-collection-raccoon-secret-society-154509442.html, 2021. Last accessed 2 november 2022.

[36] B. Edgington. *A technical handbook on Ethereum's move to proof of stake and beyond.* 2023.

[37] J. D. Eisenberg and A. Bellamy-Royds. *SVG essentials: Producing scalable vector graphics with XML.* " O'Reilly Media, Inc.", 2014.

[38] B. D. Flury. Acceptance–rejection sampling made easy. *Siam Review*, 32(3):474–476, 1990.

[39] FrankNFT-labs. Erc721f. https://github.com/FrankNFT-labs/ERC721F, 2023.

[40] Y. Gupta and J. Kumar. Identifying security risks in nft platforms. *arXiv preprint arXiv:2204.01487*, 2022.

[41] T. Hepp, M. Sharinghousen, P. Ehret, A. Schoenhals, and B. Gipp. On-chain vs. off-chain storage for supply-and blockchain integration. *it-Information Technology*, 60(5-6):283–291, 2018.

[42] J. Kauflin. Did the son of the worlds third-richest person trade nfts with inside information?, March 2022. forbes.com.

[43] M. M. A. Khan, H. M. A. Sarwar, and M. Awais. Gas consumption analysis of ethereum blockchain transactions. *Concurrency and Computation: Practice and Experience*, 34(4):e6679, 2022.

[44] D. Khazanchi, A. K. Vyas, K. K. Hiran, and S. Padmanaban. *Blockchain 3.0 for sustainable development*, volume 10. Walter de Gruyter GmbH & Co KG, 2021.

[45] D. E. Knuth. *Seminumerical algorithms.* The art of computer programming 2. Addison-Wesley, Reading (Mass.), 2nd ed. edition, 1971.

[46] P. Kostamis, A. Sendros, and P. Efraimidis. Exploring ethereum's data stores: A cost and performance comparison. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 53–60. IEEE, 2021.

[47] Y. Labs. Bored ape yach club collection. URL: https://opensea.io/collection/boredapeyachtclub, 2021. Last accessed 2 november 2022.

[48] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15. IEEE, 2020.

[49] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15. IEEE, 2020.

[50] T. Nguyen-Van, T. Nguyen-Anh, T.-D. Le, M.-P. Nguyen-Ho, T. Nguyen-Van, N.-Q. Le, and K. Nguyen-An. Scalable distributed random number generation based on homomorphic encryption. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 572–579. IEEE, 2019.

[51] E. Reguerra. Nfts minted on ftx break highlighting web2 hosting flaws. URL: https://cointelegraph.com/news/nfts-minted-on-ftx-break-highlighting-web2-hosting-flaws, 2022. Last accessed 24 May 2022.

[52] A. Reutov. Predicting random numbers in ethereum smart contracts. 2018.

[53] K. Sigman. Inverse transform method. *Class notes,[Retrieved 2019-11-11] Available at: http://www. columbia. edu/ ks20/4404-Sigman/4404-Notes-ITM. pdf*, 2010.

[54] J. J. Soria Ruiz-Ogarrio et al. Mining incentives in proof-of-work blockchain protocols. *Publications of the Faculty of Social Sciences/Department of Political and Economic Studies, University of Helsinki*, 2022.

[55] D. Stansbury. Rejection sampling, 2014.

[56] J. M. Steele. Non-uniform random variate generation (luc devroye), 1987.

[57] S. Vimal and S. Srivatsa. A new cluster p2p file sharing system based on ipfs and blockchain technology. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–7, 2019.

[58] Q. Wang, R. Li, Q. Wang, and S. Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447*, 2021.

[59] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger.

[60] S. H. Yudanto. [solidity] contract code size over limit. 2021.

[61] G. Zheng, L. Gao, L. Huang, and J. Guan. *Ethereum smart contract development in solidity*. Springer, 2018.

[62] C. Zhu. Nft sneaker marketplace design, testing, and challenges. 2022.