Imperial College London

INDIVIDUAL PROJECT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Exploring Security & Gas Usage Of Smart Contracts Through P2Es

Author:
Alex Richardson

Supervisor:
Naranker Dulay

Tuesday 20th June, 2022

Submitted in partial fulfillment of the requirements for the BEng of Imperial College London

Abstract

The crypto space is a novel area, and despite the large sums dealt with, many projects do not follow good development practices. Subsequently, this means that users who interact with such protocols are at risk of losing funds through security vulnerabilities or wasting money on gas through inefficient code. To combat this, the aim was to find the optimal design patterns, procedures and tools that all developers should be using. Furthermore, solutions to programmatically verify the security of smart contracts and off-chain alternatives to reduce gas costs were presented. In addition, some of the most popular protocols, such as Axie Infinity, were analysed to discover best practices for development.

Using my findings, a decentralised Play to Earn game, Olympus, was developed to learn first-hand and illustrate a good framework for building Web3 projects. In making Olympus, it was found that automating the test suites and multiple tools, using varying techniques, such as static analysis and fuzzing, was the most beneficial in identifying bugs. To reduce the cost of computation, an alternative to a common price feed utility, Chainlink, was my own price prediction model. Additionally, solutions such as variable packing and carefully choosing data structures were employed to reduce gas consumption in development. However, it was found that the most significant gas savings were possible by moving the code off-chain at the sacrifice of decentralisation.

Acknowledgments

I would like to thank my mum, family and friends as I would not be where I am today without any of them.

Contents

Chapter 1

Introduction

1.1 History

A decentralised currency [1], called Bitcoin, was implemented by Satoshi Nakamoto in 2009. This decentralised currency uses a Proof-Of-Work consensus, which is important because it allows a peer-to-peer network to collectively agree on transactions and allows free entry into the process.

The implementation and popularity of Bitcoin [1] gave birth to multiple alternative projects, most importantly (in regards to this project) Ethereum. Ethereum is an alternative project that allows the creation of decentralised applications without the need for a new blockchain.

1.2 P2E

Play-To-Earn (P2E) [2] is a term for games that allow players to earn cryptocurrency for playing. P2E games are revolutionary because they allow players to have complete ownership of their in-game assets, which can be bought, sold, or traded in the open market and within the game, opening up many possibilities for gaming. The assets the players own have a real-world value, so P2E games propose a method for gamers to increase the value of their assets through gameplay, making gaming more desirable.

1.3 Motivation

There is a need for blockchain technology in today's society which is becoming more and more apparent each day. There are many real-world problems that decentralised technology is solving [3] such as centralised authority, single point of failures, and censorship. Firstly, people do not need to trust a central authority. More and more we see central authorities abusing people's trust every day, so having a trustless network can give people peace of mind. Secondly, with decentralisation, there is no single point of failure. Therefore, outages of a decentralised network are unlikely

and people have more of a guarantee that they will not be let down. Thirdly, central authorities are commonly known for censoring content that people can access. However, with a decentralised network, it is increasingly difficult to censor content.

A major issue that has become more prevalent is the number of exploits in these blockchain applications, directly affecting users' wealth via cryptocurrency assets. An example is the recent Axie Infinity security vulnerability [4] which led to hackers stealing $650M from the blockchain P2E game. There are a large number of common bugs which developers are not picking up due to the relatively new tools and technologies they are using.

Another concern with blockchain applications is the need to push out code fast without checks for inefficiencies in the contract. This leads to an overabundance of applications with functionality that require high computational cost which leads to high gas costs. This issue is particularly experienced by Ethereum users where costs of gas may exceed the value of the computation itself.

The joining of cryptocurrencies with games is in its early stages which makes it a prime target for malicious actors. Inexperienced developers have minimal examples of best practices and make their applications susceptible to devastating attacks. Therefore, this project can be used as an opportunity to explore multiple methods used to make a P2E blockchain game secure and minimise its gas consumption.

1.4 Contributions

It is planned to create a P2E game that anyone can play using a Web3 wallet. The main challenges will be to introduce key features for the game while also ensuring the security of the smart contract code. This is because malicious users can take advantage of vulnerabilities in smart contracts and potentially capitalise on ill-gotten gains from them. Additionally, the efficiency of the smart contracts will be a priority because it will directly influence the transaction fees for players.

Increased engagement with the application for users requires advanced features that reward players who are often actively playing. Such features would need to provide incentives to encourage play and not penalise players who withdraw earnings. With earnings being distributed through a pull payment style dynamic airdrop it should significantly reduce transaction fees incurred by the player. A decentralised finance service, i.e. a system to carry out financial transactions on the blockchain such as lending game assets between players, will improve the player's experience by introducing a possible income stream from interest while playing the game.

The project will be a reference point for future blockchain developers and will showcase a novel combination of industry-leading techniques to develop efficient and secure applications. Anyone interested in developing a P2E will be able to learn from the discoveries throughout this project and apply them to their own projects.

The following will be the main focus of the project:

1. Mitigating potential security risks and vulnerabilities in smart contracts
2. Performing code optimisations to reduce gas consumption
3. Develop in-demand blockchain services such as DeFi and dynamic airdrops
4. NFT price prediction model instead of standard off-chain price feeds
5. Implement custom scripts to maintain the integrity of code

1.5 Ethical Issues

Since the project is cryptocurrency-related, one ethical issue is the potential damage to the environment, as the process of mining requires a substantial amount of energy. However, Brain Brooks said [5] "cryptocurrency mining used 58% of sustainably sourced power compared to 31% for the US economy as a whole". Although cryptocurrency has a high energy consumption, this does not guarantee that it damages the environment.

The project will track players' Web3 wallet addresses. However, this information cannot be used to uniquely identify anyone, so I believe this is not an ethical issue.

Since it is planned to collect information from a focus group, ethical issues such as privacy and data protection are involved. I plan to respect the privacy of each member of the focus group and withdraw or anonymise any information they have given if desired.

Chapter 2

Blockchain

A blockchain [6] is a public database that is shared across multiple computers in a network. Data and state are stored in a block and each block cryptographically references its parent, so the blocks are chained together. Since the data in a block is immutable, someone wanting to change the data in a block would have to change the block itself to alter the data. However, to change a block all following blocks must be changed as well, which requires the entire network to agree upon.

2.1 Ethereum

Ethereum [1] is a project that allows people to easily build decentralised applications by abstracting the blockchain. The Ethereum blockchain has a built-in Turing-complete programming language, allowing people to write their smart contracts.

2.1.1 Ethereum Blockchain

The Ethereum blockchain [1] has its differences from the Bitcoin blockchain, however, they are mostly similar. The most contrasting difference is that the Ethereum blockchain contains a copy of the transaction list and the most recent state. One feature all blockchains should have in common is that every block must be verified before adding it to the blockchain; a simplified overview of the Ethereum block validation algorithm is:

1. Validate previous (parent) block referenced
2. Check if current timestamp is after the previous timestamp
3. Validate multiple Ethereum-specific concepts are valid (e.g. difficulty)
4. Validate the PoW on the block
5. Check for errors and insufficient gas
6. Validate the final state

The state [1] is stored in a tree structure, that is stored in each block. Initial thoughts may suggest that this is inefficient however, between each block only a minuscule

section of the tree is changed. Thus, the data that has not changed is referenced using pointers to prevent storage duplication. Due to this implementation, storing the whole blockchain history is redundant - this would save Bitcoin up to 20 times of the space used.

2.1.2 Proof-Of-Work

Proof-Of-Work [7] is the consensus protocol used by Ethereum, and also Bitcoin. As mentioned previously, PoW allows the nodes of the Ethereum, and Bitcoin, network to agree on the state of information on the blockchain. This is momentous for the security of any cryptocurrency's network because it makes it more difficult for users to "double-spend" their coins or tokens in addition to other malicious actions, such as maintaining a second chain, fabricating blocks, or creating new blocks that erase transactions. For a malicious user to consistently carry these actions out they would need over 51% of the network mining power to triumph over everyone else.

One purpose of PoW [7] is to extend the length of the blockchain because the longest blockchain will have had the most computational resources spent on it, making it the most believable one.

One of the main criticisms of PoW [7] is that it requires the same energy of a medium-sized country, like Austria, to uphold the security and decentralisation of the Ethereum network.

2.1.3 Mining

Cryptocurrency networks [8] need to ensure that everyone agrees on the transactions. Miners use the PoW consensus protocol (solving computationally difficult puzzles) to secure the network while being rewarded with two new ETH and all transaction fees (gas). Anyone with a computer, and mining software, can mine on the Ethereum network. However, to mine ETH for a profit, specific computer hardware is required.

PoW [7] sets the difficulty and the rules for the miners. The miners undergo a trial and error race to find a valid hash (length influenced by the difficulty) to use for the block to add. The previous blocks in the chain are used to calculate the hash, so if one transaction were to change, the hash would be completely different, indicating fraud.

2.1.4 Proof-Of-Stake

PoW [7, 9] will be replaced by a better consensus algorithm on the Ethereum network, called Proof-Of-Stake (PoS) to support the scalability of the network. Consequently, mining will also be brought to an end.

PoS [9] will replace miners with validators, which anyone can become by staking their ETH. Although validators are replacing miners, they have the same responsibilities of ensuring all the nodes agree on the state of the network but instead of miners racing to solve a computation, the validators are randomly chosen to create blocks and check blocks others have made.

The reason PoS [9] has been planned to replace PoW is that PoS offers many improvements over PoW. Firstly, the removal of miners significantly reduces the energy used to validate transactions, making the network more eco-friendly. Likewise, removing miners lowers the entry barrier since specific high-spec computer hardware is now unnecessary. Thirdly, a lower entry barrier should promote an increase in the number of validators - strengthening the decentralisation of the network, which is one of the core reasons for cryptocurrency's popularity. Lastly, staking ETH is a method to encourage good behaviour as a validator because their staked ETH is at risk. Bad validator behaviour (failing to validate or connivance) can result in the loss of capital by having their staked ETH slashed.

Similar to PoW, PoS [9] is also vulnerable to a 51% attack. However, with PoS there is additional risk, for the attackers, because the attackers must have control over 51% of the staked ETH, which is an astronomical amount of money.

2.1.5 Gas

Gas [10] is the unit of measure for the computational effort to execute operations on the Ethereum network. Every Ethereum transaction requires computational resources, so a fee is mandatory for each transaction. Thus, gas frequently refers to the fee required for a successful transaction on the Ethereum network.

Every [10] block has a base fee, derived from the demand for the block space, which is burnt. Therefore, users of the network are expected to set a tip (priority fee) to reimburse miners for adding their transactions in blocks. Consequently, transactions with a higher tip will presumably be executed before all others since the miners will be rewarded more for finding the corresponding hash.

Gas is a necessity [10] because it maintains the security of the Ethereum network by acting as a deterrent for spam. There is also a maximum gas limit to prevent accidental or malicious infinite loops, in addition to any other computations that are made to waste resources.

One significant issue with gas [10] on the Ethereum network is extremely expensive gas fees at times of high demand, which forces the users to offer a higher tip, otherwise their transaction is unlikely to be put into the next block. The high gas fees are dominantly caused by the popularity of Ethereum and the complexity of decentralised applications and smart contracts.

2.1.6 EVM

The Ethereum Virtual Machine (EVM) [11] defines the rules for computing a new valid state from block to block and is maintained by a colossal number of connected computers running an Ethereum client. Moreover, it is the environment all smart contracts and Ethereum accounts live.

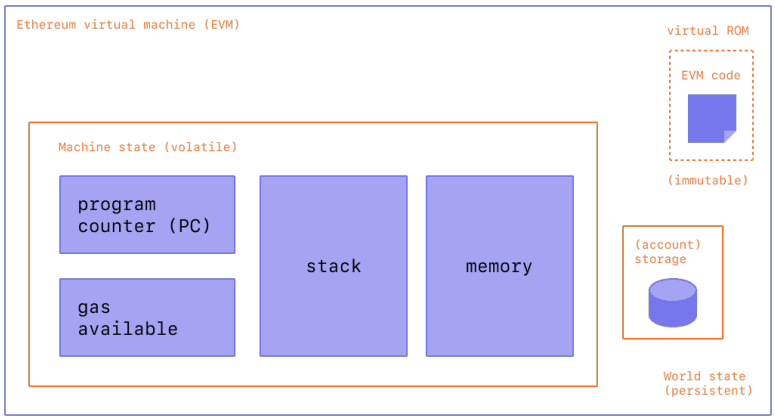

Figure 2.1: EVM illustrated [11]

2.1.7 Solidity

Solidity [12, 13] is a statically-typed, high-level language for smart contract development on Ethereum, with consistent bi-weekly releases. However, only the latest version should be used when deploying contracts for security reasons. Solidity supports inheritance, the use of libraries, events, user-defined types and many more features for an easy-to-use experience.

2.1.8 Smart Contracts

Smart contracts [14] are a public program that runs on the Ethereum blockchain, that anyone can write. Smart contracts are deployed on the Ethereum network, costing more gas than usual, and can be interacted with by executing the defined functions in the contract, which are irreversible. Additionally, they can define rules that are automatically enforced by the code.

Smart contracts [15] are an attractive target to attack for profit because the code cannot be changed. So, there are no security patches once deployed, assets are irrecoverable, and stolen assets are difficult to track. Attackers can profit by exploiting vulnerabilities in the smart contracts or unexpected behaviour in Ethereum. Multiple bugs and vulnerabilities [16, 17] have caused funds to be lost in numerous projects in excess of $300M.

Re-entrancy

Re-entrancy [15] is the most significant security issue for smart contract development. This issue stems from the functionality of the EVM with multiple contracts executing. A contract calling another contract pauses the calling contract's execution and memory state until the call returns, which opens the opportunity for attackers to potentially steal funds by using the vulnerability known as re-entrancy. As an example, the `Victim` contract below has a re-entrancy vulnerability.

```
// THIS CONTRACT HAS AN INTENTIONAL VULNERABILITY, DO NOT COPY

contract Victim {
  // mapping of all balances deposited
  mapping(address => uint256) public balances;

  function deposit() external payable {
    balances[msg.sender] += msg.value;
  }

  function withdraw() external {
    uint256 amount = balances[msg.sender];
    (bool sent, ) = msg.sender.call{ value: amount }("");
    require(sent, "[Victim]: Failed to send.");
    // XXX: the state variable is updated after the external call
    balances[msg.sender] = 0;
  }
}
```

Figure 2.2: Victim of re-entrancy, adapted from [15]

When a user calls `withdraw()` (as shown in 2.2) the following occurs [15]:

1. Read the balance of the user
2. Send them the balance amount in ETH
3. Reset their balance to zero

If the `withdraw()` function [15] is simply called from a regular account, the expected behaviour occurs. However, if a smart contract calls `withdraw()` then `msg.sender.call{ value: amount }("")` will not only send the ETH, it will also implicitly call the `fallback` function in contract to begin executing code.

The following is an example of a malicious contract exploiting the re-entrancy bug in the `Victim` contract 2.2.

```
contract Attacker {
  address public VICTIM_ADDRESS;
  Victim public victim = Victim(VICTIM_ADDRESS);

  function beginAttack() external payable {
    victim.deposit{ value: 1 ether }();
    victim.withdraw();
  }

  fallback() external payable {
    if (address(victim).balance > 0 ether) {
      victim.withdraw();
    }
  }
}
```

Figure 2.3: Attacker exploiting re-entrancy, adapted from [15]

If the malicious `Attacker` contract calls the `beginAttack()` function the following will occur [15]:

1. `Attacker.beginAttack()` deposits 1 ETH into `Victim`
2. `Attacker` calls `Victim.withdraw()`
3. `Victim` reads `balances[msg.sender]`
4. `Victim` sends ETH to `Attacker` (implicitly executing the `fallback` function)
5. Go back to step 2 until `Victim` has no ETH left

Thankfully, the solution to the re-entrancy vulnerability [15] is very simple. Updating the state variables before any external calls removes the possibility of a re-entrancy attack by a malicious smart contract, as shown below.

```
contract NoLongerAVictim is Victim {
  function withdraw() external override {
    uint256 amount = balances[msg.sender];
    // XXX: the state variable is updated before the external call
    balances[msg.sender] = 0;
    (bool sent, ) = msg.sender.call{ value: amount }("");
    require(sent, "[Victim]: Failed to send.");
  }
}
```

Figure 2.4: No longer a victim of re-entrancy, adapted from [15]

Now, if the Attacker contract calls the `beginAttack()` function the following will occur [15]:

1. `Attacker.beginAttack()` deposits 1 ETH into Victim
2. `Attacker` calls `Victim.withdraw()`
3. `Victim` reads `balances[msg.sender]`
4. `Victim` updates `balances[msg.sender]` to 0
5. `Victim` sends ETH to `Attacker` (implicitly executing the fallback function)
6. `Attacker` calls `Victim.withdraw()`
7. `Victim` sends `Attacker` 0 ETH

2.1.9 Security Tools

Re-entrancy [15, 18, 19] is one vulnerability of many. Fortunately, there are multiple security tools available to use to locate numerous vulnerabilities in smart contracts to help prevent funds from being stolen or lost. Firstly, Slither is a static analyser that simultaneously approximates and analyses all paths of the program. Secondly, Echidna executes the code with a pseudo-random generation of transactions attempting to violate any given property. Thirdly, Mythril uses symbolic execution in addition to other techniques to detect security vulnerabilities. These are just a few among many security tools available for the community to use.

Tool	Technique	Duration	Vulnerabilities Missed	False Positives
Slither	Static Analysis	Seconds	Moderate	Low
Echidna	Fuzzing	Minutes	Low	None
Mythril	Symbolic Execution	Minutes	None *	None

Table 2.1: Security tools information [18]

* if all paths are explored

2.1.10 OpenZeppelin

OpenZeppelin [20] is a popular open source library consisting of secure smart contracts, with many token standards and utilities available to use. The OpenZeppelin library code has been vetted by the community and is the preferred method for creating smart contracts for decentralised projects.

ERC-20

Tokens can represent a lot in Ethereum, [21] such as experience points in a game, skills unlocked for a character in a game, or a fiat currency like USD. Since tokens can represent virtually anything, the ERC-20 standard was introduced to allow developers to construct fungible tokens that are compatible with other services. Any token

using this standard is a fungible token. OpenZeppelin [20] provides the ERC20.sol contract which implements the ERC-20 token standard.

SafeERC20

SafeERC20 [20] is a wrapper for ERC-20 token implementations, which provides functions, such as `safeTransfer()` that revert on failure so contracts can securely interact with the tokens. The comparison of interacting with an ERC-20 and SafeERC20 is shown below.

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract FreeTokens {
  using SafeERC20 for IERC20;
  IERC20 private immutable _token;

  function withdraw(uint256 amount) external {
    // OLD
    require(_token.transfer(msg.sender, amount),
      "[FreeTokens]: Failed transaction"
    );
    // NEW
    _token.safeTransfer(msg.sender, amount);
  }
}
```

Figure 2.5: ERC-20 token transfer comparison, inspired from [20]

Ownable.sol

The contract Ownable.sol [20] provides a fundamental form of access control. Without access control in a contract, anyone could mint tokens indefinitely, self-destruct the contract, or vote on proposals in a DAO. By default, the address that deploys the contract is the owner. Typically, this person will regulate the project by executing the administrative functions available if needed, such as minting and burning tokens.

ERC-721

In addition to fungible tokens [22], non-fungible tokens also have their own standard - ERC-721. Every non-fungible token using this standard has a `uint256 tokenId` variable, which can be used to uniquely identify an object. OpenZeppelin [20] provides a contract that implements the ERC-721 token standard.

ERC-721A

ERC-721A [23] is an improvement of the ERC-721 standard, by Azuki, with a focus on reducing gas for multiple mints. The entire Ethereum ecosystem has to pay extreme gas prices when popular NFT projects start to mint. Therefore, reducing gas fees on the Ethereum network should be a top priority for user satisfaction.

Azuki have focused on optimising functions from the ERC721Enumerable contract. Firstly, removing duplicate storage in each NFT's metadata. Secondly, the balance of the owner is updated only once per batch mint instead of multiple times per minted NFT. Thirdly, the owner data is only updated once per batch mint instead of multiple times per minted NFT. From all these optimisations, Azuki's ERC-721A contract saves up to 531708 gas compared to OpenZeppelin's ERC721Enumerable contract.

ERC-721R

A rug pull [24] usually consists of a project gaining significant popularity in the early stages, then once the tokens are released the creators abandon the project making the tokens worthless. Rug pulls are rampant in the crypto ecosystem due to the lack of regulation.

The ERC-721R [25, 26] is based on the ERC-721A standard, but allows for trustless refunds within 30 days. This prevents rug pulls, selling below floor price, and reduces the risk significantly.

In conclusion, the ERC721A standard [23] significantly improves upon the OpenZeppelin contract in terms of gas consumption. The ERC721R [25, 26] standard has the gas consumption improvements from the ERC721A contract and allows for refunds which greatly increases the confidence of safety in the buyer. However, OpenZeppelin is the most popular smart contract library in the community and has been audited on many occasions, making it the most secure choice out of the contracts to choose from. Thus, ERC-721 has been chosen for the NFTs of Olympus.

Governor.sol

Initially, a centralised authority [20] is usually in control of the development of cryptocurrency projects to ensure that it receives constant updates to improve. However, this centralised authority impedes the decentralisation of such projects, so on-chain governance is eventually used to allow a community of stakeholders to vote on changes such as contract upgrades and treasury management.

The Governor.sol contract [20] provides the functionality for on-chain governance with customisable features such as, voting delay, voting period and a proposal threshold. These parameters are defined in terms of blocks instead of time, to prevent time manipulation from block miners.

Pausable.sol

Pausable.sol [20] provides a common emergency response that can pause functionality by using the modifiers whenNotPaused and whenPaused, and the inherited functions. Typically, the creators of a project work towards a solution while the contracts are paused due to an emergency.

ReentrancyGuard.sol

The ReentrancyGuard.sol [20] contract provides functionality to prevent reentrant calls to functions by using the modifier nonReentrant. The functions that have the nonReentrant modifier cannot call one another (directly or indirectly).

2.2 NFTs

A fungible asset [27] is something that can be interchanged for the same value. For example, you can interchange a £20 note for two £10 notes for the same value. However, you cannot swap a non-fungible asset for the same value because each non-fungible asset has unique properties.

Non-fungible tokens (NFTs) [27] are unique digital assets that are popular due to their use with digital artwork. Using NFTs, one can create a digital certificate of ownership for artwork that can be bought or sold. Moreover, a royalty fee for each time the NFT is sold can be programmed into a smart contract for the original artist.

NFTs have been surrounded by a lot of speculation and can prompt people to consider why some NFTs are worth a great deal of money such as [28] the first tweet of Jack Dorsey, Twitter co-founder, with bids reaching as high as $2.5M. However, there are [29] limitless possibilities for NFT use cases, including games. Skins are visual enhancements that usually personalise gaming avatars, which players often pay for. It is not uncommon for players to spend a substantial amount of money on skins that provide no enhancement to the gameplay, only the appearance of the game. These skins are the equivalent of digital art for games, and NFTs can allow ownership and authentication of these skins. NFTs can also provide transferability of capital between games. Usually, once a player has spent money on a game for a skin, that skin can only be used within the game it was purchased. However, with NFTs, a player can sell an unwanted NFT of a skin for one game and purchase another for a different game. This area is newly founded and there are still many many possibilities for NFTs within the gaming industry to be discovered.

2.3 DAOs

Decentralized Autonomous Organizations (DAOs) [30, 31] are software-enabled organisations that are owned and managed by their members without centralised leadership. DAOs have a treasury pool that can only be accessed by people with the approval of the group that owns the DAO. DAOs are trustless, transparent and verifiable by anyone. Therefore, no misconduct cannot go unnoticed unlike with centralised organisations.

A smart contract [31] defines the rules of the organisation in addition to storing the DAO's treasury. Thus, decisions will be automatically authorised, by the smart contract, once the group makes a collective vote. The smart contract code is tamper-proof once deployed, so any actions without the group's collective approval will automatically fail, ensuring the safety of the treasury. Hence, there is no need for a centralised authority in a DAO.

2.4 DeFi

Decentralised finance (DeFi) [32] is financial services on blockchain such as, lending, borrowing and trading without any third party involved. DeFi creates opportunities for people who would not be allowed to use finance services with traditional finance.

2.4.1 Lending

People often lend their crypto to earn interest because the interest rates are extremely alluring (compared to traditional finance). Neither party involved in decentralised lending [33] has to identify themselves. Therefore, the borrower must put up collateral of value that exceeds the assets borrowed. Otherwise, the borrower could borrow and sell the assets for a profit.

2.4.2 Liquidation

Since neither party [33] has to identify themselves, people can borrow without a credit check or sharing personal information. However, this may lead to borrowed assets never being returned. In these situations the lender is entitled to liquidate the borrower and claim the total collateral.

Chainlink

Loaned assets with a variable price can fluctuate by a significant amount. So, if the price increases above the value of the collateral it puts the lender at risk they should be allowed to liquidate the borrower. However, for this to happen the smart contracts need access to off-chain data.

Hybrid smart contracts [34] combine on-chain code with off-chain code provided by Decentralised Oracle Networks (DONs). Hybrid contracts enable functionality with tamper-proof and immutable properties in addition to, access to real-world data. Furthermore, DONs provide a high level of tamper-resistant and reliability in an isolated off-chain environment using a variety of security approaches to achieve mainly fetching, validating, securing and delivering data from external APIs and perform various computations for smart contracts. DONs can be used to fetch price feeds, so can be used for liquidation functionality in smart contracts.

2.5 Solana

Currently, Solana [35] is among the fastest blockchains (400 millisecond block times), with a focus on scalability ensuring its speed and low transaction fees are maintained as the ecosystem grows. Furthermore, Solana has thousands of independent nodes to ensure decentralisation and maintain the safety of transactions.

Solana [36] smart contracts are called programs. The programs deployed on-chain and run via the Solana Runtime are written in Rust, C, and C++. Solana maintains two sets of programs that are part of the core software releases: Solana Program Library (SPL) and Native programs. Native programs are the bedrock of Solana that provide many fundamental features of a cryptocurrency ecosystem such as, creating an account, transferring SOL (Solana's native token), staking and voting. On the contrary, the SPL includes the Token program which is the equivalent of the Ethereum ERC-20 standard that allows you to mint, transfer, burn or query tokens.

2.6 Cardano

Cardano [37] is a decentralised PoS blockchain platform for the ADA cryptocurrency, with a focus on security, scalability, sustainability and transparency. Since security is an important core principle of Cardano, Haskell was chosen as the implementation language. This allows for many benefits, such as testing components in isolation, property-based testing, and running tests in simulations to ensure code correctness.

2.6.1 Plutus

Cardano's [37] native smart contract language is a Turing-complete language written in Haskell, called Plutus. Plutus was inspired from modern language research to provide extensive security advantages, so you can be confident in the correctness of the smart contract execution. Additionally, Plutus interacts with on-chain and off-chain code using the Plutus compiler and the Glasgow Haskell Compiler (GHC) respectively. So, both code are based on the same language, offering an easy-to-use uniform code base.

Chapter 3

P2E Games

In this chapter, a few of the most popular P2E games are explored in detail to understand the underlying game mechanics and how they function with blockchain technology. Additionally, games that could potentially be made into P2E games were discussed to identify key features that allow for interesting combinations.

3.1 Existing P2E Games

3.1.1 Axie Infinity

Axie Infinity [38] is a digital pet P2E game where players trade, battle and raise fantasy creatures called Axies. Axie was the first blockchain game to introduce earning for playing the game and also the first mobile blockchain game. To play the game, people only need to download a Web3 wallet, such as MetaMask, and own three Axies.

Axie Infinity [39] has a feature to breed Axies, with the cost always below the floor price for Axies. However, there are additional requirements to breed, such as Love Potions (earned by playing the game), to prevent spamming. These requirements incentivise players to play the game as they are given a clear objective and a reward after completion.

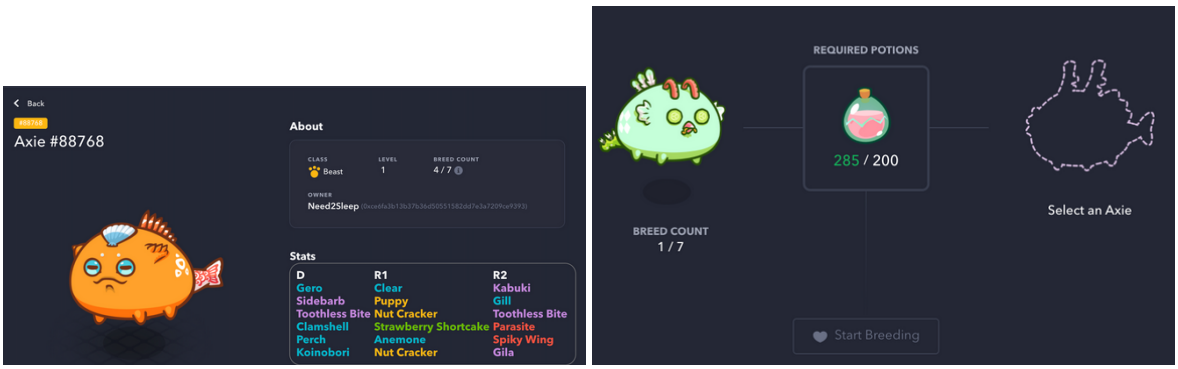

Figure 3.1: Axie on the marketplace, [40]

3.1.2 Upland

Upland [41] is a property trading P2E game with a native token called UPX and players having companions called “Block Explorers” as a core mechanic. Every player is initially given a Block Explorer since these are required to progress in the game, in addition to 3000 UPX as a generous beginner present. Players can buy, sell and trade properties using UPX, as well as completing “Collections” to boost their earnings from these properties.

An unusual feature of Upland [41] is that only players who have accumulated 10000 UPX earn the title “Uplander”, allowing them permanent ownership of their digital assets within the game. Furthermore, if players have not yet achieved the title “Uplander” they must go on the game weekly, otherwise, their digital assets are redistributed into Upland’s economy. A shortcoming of this game design is if the “Uplander” title is too difficult, then this can initially deter players from starting the game; negatively impacting the game’s ecosystem, showing the potential impact of poor game design.

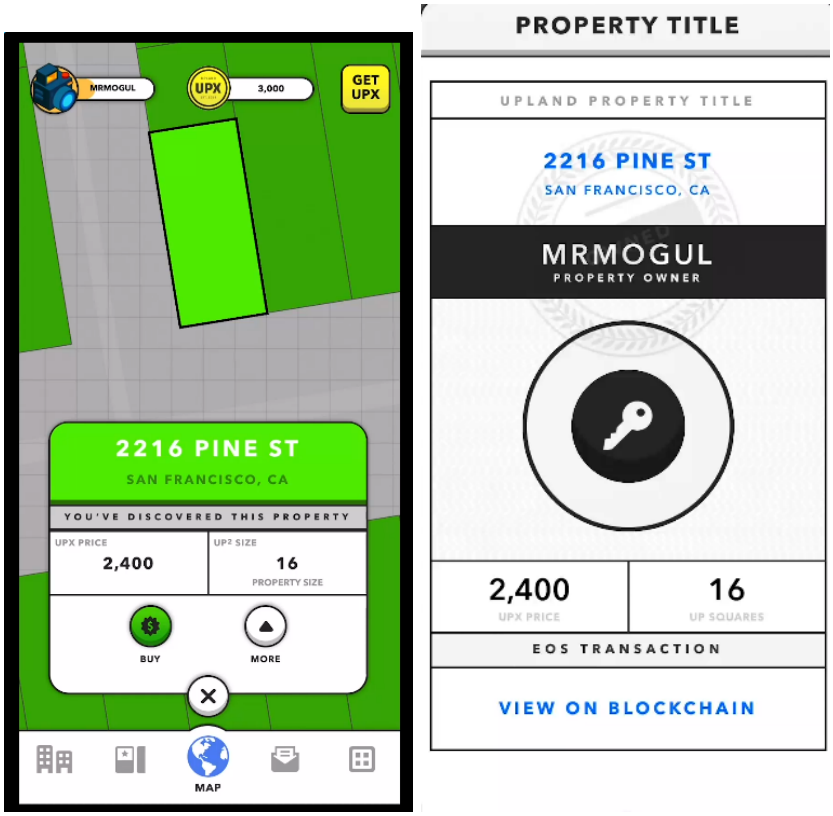

Figure 3.2: Upland property title, [42]

3.1.3 Alien Worlds

Alien Worlds [43, 44] is an NFT DeFi metaverse, with games, where players can earn in-game currency called Trilium (TLM). Within the metaverse, there are several planets, that represent DAOs, which receive Trilium from the metaverse smart contract daily. Players can mine, with NFT tools, on any given planet to earn Trilium or a newly minted NFT. An NFT tool is a requirement to mine but every new player is given a standard shovel NFT, removing the entry barrier to play. Alternatively, players can stake their Trilium on a planet to earn voting rights for that DAO to elect Planetary Councilors. Due to all the captivating features, Alien Worlds Mining has become one of the most popular blockchain games, with an average of more than 10 million transactions/plays per day.

Figure 3.3: Planet DAOs, [45]

3.2 Games with P2E Potential

Before making a P2E, it was explored if already existing games would make good P2Es. Games that already involve money or unique items would have a high affinity to be a P2E but other games could prove to be compatible.

3.2.1 Monopoly

Initial thoughts would suggest that Monopoly would make a perfect fit for a P2E game. Players can play with a cryptocurrency token instead of the usual paper money. There would be no need for a player to handle the bank's money or the auctions, because the smart contracts can automate all of the respective functionality. Furthermore, the property cards can be turned into NFTs since each property card is unique, with different rent values.

Each player is given a certain amount of money at the start of a game, so either the contract would have to lend tokens or players would have to buy the tokens to play. If the players are required to buy tokens that would create an entry barrier, which would have a negative impact on the game's community, but if the smart contracts had to distribute the tokens at the start of each game, there may not be enough liquidity. Additionally, in a game of Monopoly the bank "never goes broke" but in a P2E this would require an infinite or sufficient amount of liquidity for each concurrent game, which is impractical.

3.2.2 Poker

Poker, or any betting card game, is a great fit for a P2E. Instead of betting money, users can bet cryptocurrency. For games with a dealer, smart contracts can replace them. However, the disadvantage of using smart contracts is that there is a potential for security vulnerabilities and a betting cryptocurrency game would be alluring for any attacker.

3.2.3 Chess

Chess may seem like an unusual game to combine with cryptocurrency but players could be rewarded with a native token that they could spend to purchase NFTs of skins for the chess pieces. Games usually offer skins that only change the cosmetic of the character, or chess pieces in this instance, that offer no advantage in the game, so this idea is not unconventional as it may seem.

3.3 Conclusion

The research above provides an insight into what elements are used to construct popular P2E games currently. It is clear that a central aspect of a P2E game consists of tradeable assets that are used to enhance a player's experience while playing. Customisation and integrated marketplaces seem to offer a more immersive and engaging system with prominent games opting to implement these.

Chapter 4

Machine Learning

Machine learning will be used for the price prediction model to allow liquidations, which is discussed later on in the implementation. Since this is a regression problem, regression models and loss functions have been researched.

4.1 Regression

Regression [46] is a method used to determine the strength of correlation between one dependent variable and a series of independent variables.

4.2 Decision Trees

Decision trees [47, 48] are a non-parametric method that learn simple decision rules inferred from the dataset features to incrementally break down the dataset into smaller subsets where finally it will be made up of multiple nodes and leaf nodes for the predicted value.

4.3 Neural Networks

Artificial neural networks (ANNs) [49] are a subset of machine learning with the architecture of connected neurons similar to the human brain. Neural networks consist of an input layer, multiple hidden layers and an output layer that are all connected with an associated weight and threshold. The threshold is used to prevent nodes with a value below it from activating and passing on data to the next layer of the network.

4.3.1 Activation Functions

Activation functions are a core component of neural networks and are applied to neurons. In this paper three different activation functions are used.

Sigmoid

The sigmoid function [50] is commonly used as an activation function in neural networks for learning complex decision functions, with a domain of [-∞,+∞] and a range of [0,1]. The sigmoid function σ is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Tanh

The hyperbolic tan function is another activation function that behaves in a similar manner to the sigmoid function, but with a range of [-1,1]. The tanh function is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

ReLU

The rectified linear unit (ReLU) function [51] is lower-bounded at zero and is linear for all input values greater than zero. This function is most commonly used in regression problems, as it uses less computational power and does not have the certain issue that other activation functions are prone to. The ReLU function is defined as:

$$R(x) = max(0, x)$$

4.3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) [52] are a type of ANN which uses time series data or sequential data. In contrast to traditional ANNs, the output of RNNs depend on the prior elements within the sequence and parameters are shared across each layer of the network.

4.3.3 LSTM

RNNs [52] cannot remember previous states if they were not recent. Long short-term memory (LSTM) are a type of RNN which combats the problem of long-term dependencies in RNNs with input, output and a forget gate to control the information flow.

4.4 Loss

Loss functions are used to evaluate machine learning models, so they can learn to improve. There is a wide variety of loss functions available for each different situation.

4.4.1 MSE

Mean squared error (MSE) [53] is the mean of the squared difference between the predicted and actual values, used to evaluate regression models.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$where$$

$$n = number\ of samples$$

$$y = true\ value$$

$$\hat{y} = predicted\ value$$

4.4.2 R^2

R^2 [54], also known as the coefficient of determination, measures the overall accuracy of a regression model. R^2 is measured in the range [0,1] with a value of one meaning perfect predictions and zero meaning no predictions were of value.

4.4.3 Conclusion

Initially, a decision tree and a neural network was considered but they lacked the features for price prediction. However, the LSTM model solves this issue with its memory - providing an accurate prediction for Olympus NFTs.

Chapter 5

Proof of Concept (Magic Number)

Before developing a proof of concept, a network had to be decided. Ethereum [55] is the most popular smart contract development platform, with one of the highest market capital values. However, due to Ethereum's popularity, the network experiences congestion frequently, so transactions may not always be processed quickly. Moreover, Ethereum has had many security issues come to light in the past. According to a study [56], 34000 smart contracts developed on Ethereum had bugs that made them vulnerable.

The Solana network [57] is one of the fastest blockchains but it does not come without its faults. The network has had multiple outages and suffered from degraded performance on several occasions. Some of the outages have been due to the fact that the network has been targeted by Denial of Service attacks such as NFT minting bots going rampant on the network. Even though Solana boasts its the fastest, its uptime issues would be a severe hindrance for a P2E.

Cardano [58, 59] released the Alonzo hard fork in September 2021, which introduced the execution of smart contracts. The Cardano smart contract ecosystem is still in its early stages and having faced concurrency and scaling issues in the Alonzo testnet, there is no guarantee that the contracts for my P2E will execute as planned. Therefore, despite the security benefits, I will not develop my P2E on the Cardano network.

The following table 5.1 is an overview of the best feature of each network compared.

Network	Popularity	Security	Speed
Ethereum	√		
Cardano		√	
Solana			√

Table 5.1: Network comparison

From the research of potential blockchains for this project, it was concluded that the Ethereum network would serve the best for a successful P2E game. Widespread

access to the game is an important aspect to consider and the significant number of applications, such as exchanges and wallets that support Ethereum, will reduce the barrier to entry in playing the game. Ethereum's popularity also provides the benefit of comprehensive developer support and forums which provide solutions for common issues and security vulnerabilities. Additionally, Ethereum has a wide variety of tools to speed up development and produce cleaner code. This allows me to focus on tackling more important challenges while developing my P2E game.

A proof of concept of a decentralised application where anyone, with a MetaMask wallet, can bet ETH to guess a number between one and ten inclusive was created. After a day has passed, the game is closed and the cryptocurrency pool will be distributed proportionally to the amount bet by each winner. The smart contracts were deployed on the Ethereum Rinkeby test network, instead of the Ethereum main network, for development purposes. An extensive test suite, intertwined with continuous integration, was used to ensure the correctness of code for the smart contracts. Additionally, Slither, Echidna and Mythril were used to verify the security of the smart contracts.

5.1 Security

All previously mentioned security tools in table 2.1 were used to evaluate the security and gas usage of the smart contract for the Magic Number game.

5.1.1 Slither

The Slither tool reported vulnerabilities in the smart contract. A condensed version of the output can be found below.

```
>> MagicNumber.bet(uint256) (MagicNumber.sol#53-66) uses timestamp for
 ↪  comparisons
>> MagicNumber.claim() (MagicNumber.sol#82-97) uses timestamp for
 ↪  comparisons
Reference:
 ↪  https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

>> Low level call in MagicNumber._claim() (MagicNumber.sol#70-77)
>> Low level call in MagicNumber.claim() (MagicNumber.sol#82-97)
Reference:
 ↪  https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
```

Figure 5.1: Slither report (summarised)

Timestamp comparisons

The Slither tool reported dangerous timestamp comparisons in the `bet()` and `claim()` functions. The following contract demonstrates the vulnerabilities associated with timestamp comparisons.

```
1  // THIS CONTRACT HAS AN INTENTIONAL VULNERABILITY, DO NOT COPY
2
3  contract LuckyDip {
4    uint256 public prevTimeStamp;
5
6    constructor() payable {}
7
8    /// @notice Win ETH if the timestamp is divisible by 2
9    function dip() external payable {
10     require(msg.value == 20 ether);
11     require(block.timestamp != prevTimeStamp);
12     prevTimeStamp = block.timestamp;
13     // XXX: block.timestamp vulnerability
14     if (block.timestamp % 2 == 0) {
15       uint256 amount = address(this).balance;
16       (bool sent, ) = msg.sender.call{ value: amount }("");
17       require(sent, "[LuckyDip]: Failed to send.");
18     }
19   }
20 }
```

Figure 5.2: Contract with timestamp vulnerability, adapted from [60]

A miner can manipulate `block.timestamp` to win the ETH in the LuckyDip contract 5.2. However, these warnings can be ignored for MagicNumber.sol [61] since the integrity can be maintained with the timestamp varying up to 15 seconds.

Low level calls

The payment functions, `transfer()` and `send()` [61, 62] only forward 2300 gas to help prevent against reentrancy vulnerabilities. However, the recipient may run out of gas and the whole transaction will revert. Therefore, smart contracts should no longer depend on gas costs and use `call()` instead as it forwards all remaining gas available.

Therefore, all warnings reported by the Slither tool can be safely ignored.

5.1.2 Echidna

Echidna was used to verify multiple invariants and measure the maximum gas usage of multiple state variables and functions.

Function Name	Tests Passing
echidna_check_owner_is_not_zero	50000/50000
echidna_check_deadline_is_positive	50000/50000
echidna_check_target_is_between_one_and_ten	50000/50000
echidna_check_winningPool_is_non_negative	50000/50000
echidna_check_winningPool_is_less_than_total	50000/50000
echidna_check_total_is_non_negative	50000/50000
echidna_check_if_guessed_total_is_positive	50000/50000
echidna_check_stakes_is_non_negative	50000/50000
echidna_check_stakes_lte_total	50000/50000
echidna_check_if_correct_stakes_is_positive	50000/50000

Table 5.2: Echidna fuzzing report

State Variable / Function Name	Maximum Gas Used (Gwei)
total	437
deadline	482
stakes	2823
correct	2874
claim	2936
bet	109332

Table 5.3: Echidna gas report

5.1.3 Mythril

The Mythril tool performed many checks and found no bugs in the smart contract.

```
>> Query count: 551
>> Solver time: 436.7479577064514
>> The analysis was completed successfully. No issues were detected.
```

Figure 5.3: Mythril report

The MagicNumber was a prototype used to justify the choice of network used to develop the Olympus P2E game. Developing smart contracts on the Ethereum network provides access to a magnitude of guides and tools to aid with implementation and security that is not available on the Solana and Cardano networks.

Chapter 6

Requirements

This chapter presents the requirements set out for this project to be successful, and the planned features to meet those requirements.

The Olympus P2E game is a mix of the rock, paper, scissors game and top trumps. The same rules of rock, paper, scissors apply but a strength value is used as the deciding factor in the case of a traditional tie. All NFTs in the Olympus collection have associated metadata dictating if it's a rock, paper or scissor trait and a strength value in the range [0,100]. These NFTs are used to play the game of Olympus and as a reward for winning, players receive Drachma (DCM) tokens which can be used to open loot boxes to obtain newly minted NFTs, and purchase NFTs on the marketplace, or to borrow NFTs using the DeFi service. The following images are examples of potential NFT images.

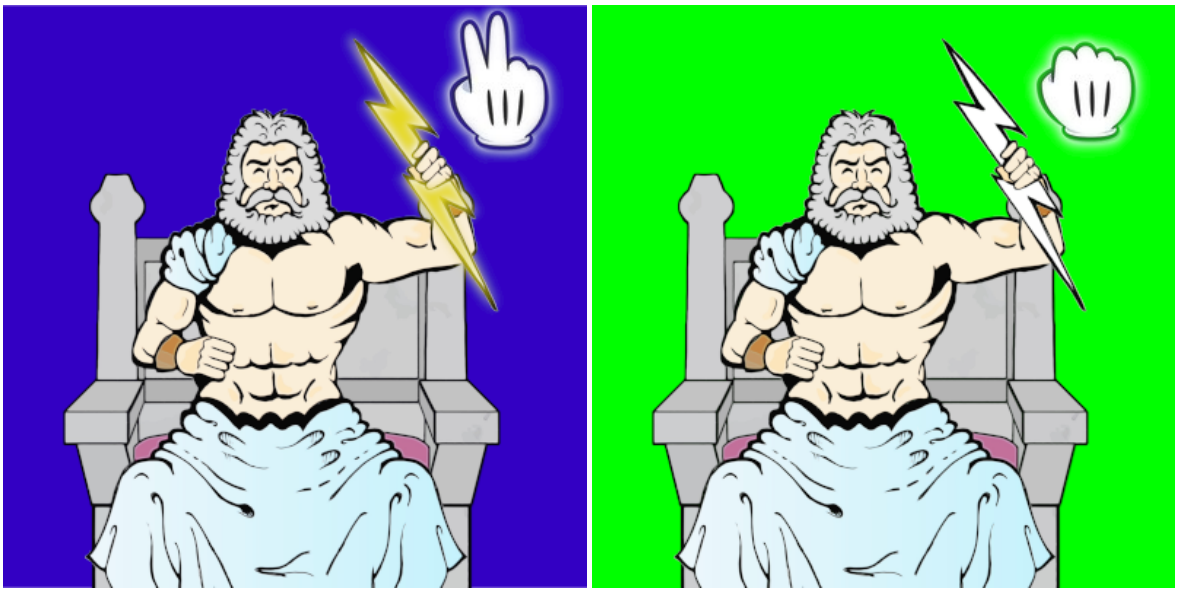

Figure 6.1: Olympus NFT images, generated using images from Freepik [63]

Goals

The main goals for this project are summarised as:

- Programmatically-verified vulnerability-free smart contracts
- Reduce gas costs lower than the most popular P2E games, such as Axie Infinity
- Replace decentralised oracles such as Chainlink with a better alternative
- Create decentralised finance services allowing lending and borrowing of NFTs

Olympus Features

The features that will be implemented to meet the goals above are:

- A decentralised reward system to pay out players with assets they have complete ownership of
- A native marketplace to buy and sell game NFT assets
- A DAO to eventually remove central authority and allow players to vote on future game changes
- Integrated DeFi service to facilitate lending out game assets between players
- Price prediction model for Olympus NFTs
- Significantly improve development procedures to reduce smart contract bugs and vulnerabilities

Chapter 7

Smart Contracts

Smart contracts are a significant portion of the project since these will dictate the majority of Olympus P2E features. While adding features only accessible through smart contracts and blockchain technology, the main focus will be to prevent introducing security bugs and lower the gas consumption of each contract.

7.1 Design

The core smart contract components are shown on a UML diagram and explained in further detail below.

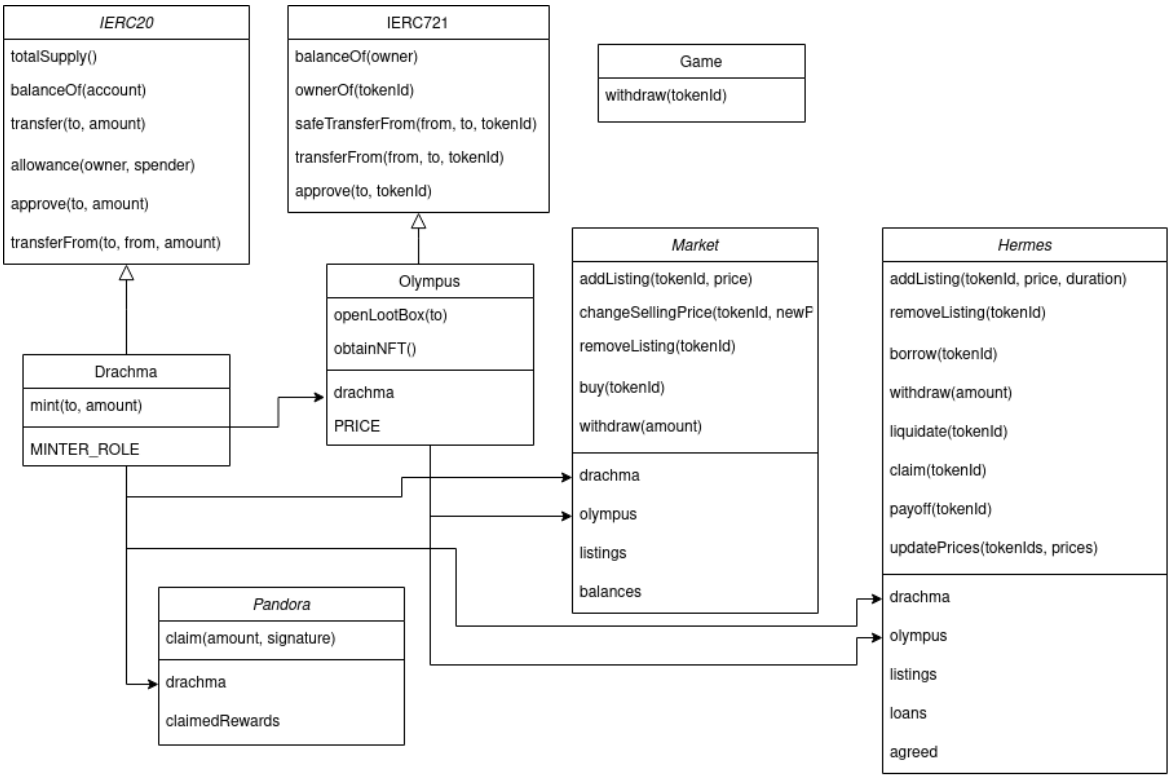

Figure 7.1: Olympus core smart contract UML

The UML diagram below shows the design to provide governance on-chain for the Olympus P2E game.

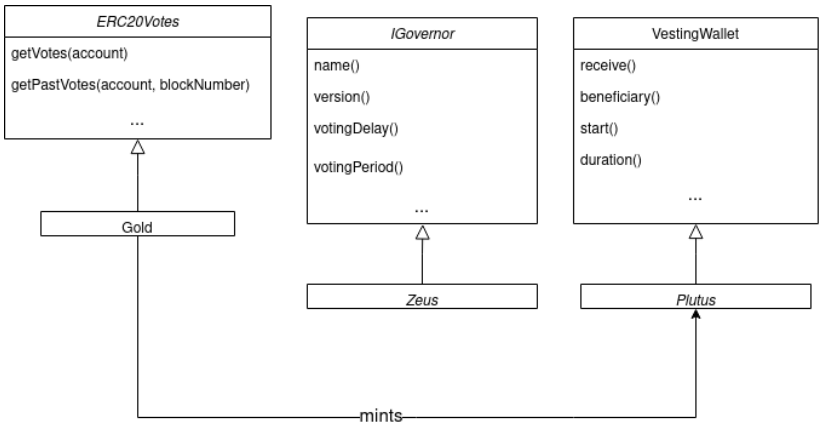

Figure 7.2: Olympus DAO UML

7.2 Drachma.sol

The native token (rewarded to players) will implement the ERC-20 standard using the ERC20.sol and AccessControl.sol contracts from the OpenZeppelin library. By using the AccessControl.sol contract, it is possible to safely implement the mint function with the administrative roles inherited.

7.2.1 Olympus.sol

The NFTs will implement the ERC-721 standard using the ERC721.sol and Counters.sol contracts from the OpenZeppelin library. The Counters.sol contract is used to set and track unique identifiers for the NFTs. NFTs can be minted with ETH or DCM.

7.3 Zeus.sol

The OpenZeppelin library [20] has been chosen for governance instead of others, such as Compound because projects with unique requirements can easily adapt the code using inheritance. Moreover, the OpenZeppelin Governor.sol contract uses less gas due to requiring a minuscule amount of storage. The Zeus.sol contract was made using the OpenZeppelin contract wizard.

The GovernorVotesQuorumFraction.sol contract [20] will be used to define the quorum, which is derived from the total supply of the token used for voting power. A quorum of 4% will be used, as this is the preferred value among DAOs. Additionally, the GovernorCountingSimple.sol contract [20] is used to provide voters with three

choices: For, Against, and Abstain. Only For and Abstain will count towards the quorum.

7.3.1 Gold.sol

The voting power for the governance of my P2E could be determined by the fungible token that players earn, so anyone who plays the game will be able to acquire voting power. However, every time a person was to play the game, their voting power would decrease. Secondly, since the token has an infinite supply, a central authority could mint an indefinite amount of tokens to increase their voting power. Therefore, a new token with a limited supply will be used for voting power.

The voting power for the governance of Olympus will be determined by the account balance of an ERC20Votes token, implemented in Gold.sol, when a proposal is put forward. The Gold.sol contract was made using the OpenZeppelin contract wizard.

7.3.2 Plutus.sol

The token used for voting power should not be available until the project has matured enough to relieve its central authority. If P2E games used on-chain governance at the start of development, updates to improve the game could potentially be voted against, preventing the game from growing. The VestingWallet.sol contract [20] manages the vesting of ERC20 tokens for a beneficiary. Plutus.sol will inherit this contract to introduce a vesting period for the Gold voting power token.

There are multiple factors [64] to consider for the vesting period. Firstly, the Gold token should be available to anyone to deter centralised authority with the majority vote power. Secondly, there should be sufficient time for the news to spread before the release of the Gold token to prevent insiders from having the advantage in acquiring a large portion of the voting power. Thirdly, the Gold token should be distributed at a uniform rate. If over half the Gold tokens are released in the first week of release, then the rest at a later date - this would allow the few early comers to hold the majority voting power, leaning towards a centralised authority within the on-chain governance.

7.4 Market.sol

An NFT marketplace is core to a P2E game such as Olympus, as this allows players to buy wanted NFTs and list unwanted NFTs to sell. A popular NFT marketplace [65] such as OpenSea or Rarible could be used to further increase the popularity of the game. For example, to list NFTs on OpenSea [66] the tokens need to implement ERC-721 and the official ERC-721 metadata standard which is an easy task to accomplish. However, nearly all prominent P2Es games create their own marketplace

for full customisability and control over additional features. Therefore, a native marketplace was implemented for the Olympus P2E game.

The Olympus integrated marketplace allows users to buy or sell NFTs easily. Users have full control over their listings and are able to remove or update them as needed to ensure peace of mind and react to possible changes in the market conditions. They are given the ability to withdraw their balance from previously sold NFTs straight to their wallets via a pull payment method which will be discussed in more detail later (7.8.1). The marketplace introduces failsafes by utilising the Pausable and Ownable modifier libraries to implement an emergency withdrawal function. In the case of an emergency that lets an attacker exploit this marketplace contract, the contract is paused to prevent withdrawals. The emergency withdrawal allows the contract owner to continue verified withdrawals under their supervision so the marketplace can continue running in a safe mode.

7.5 Hermes.sol

The DeFi platform should allow users to lend their NFTs to earn interest and borrow NFTs to use to play the game. To limit the yield of interest, players are limited to one loan term at a time.

7.5.1 Interest Rate

The lender's interest is calculated from the price of the NFT and the duration of the loan.

Let p be the price of the NFT, $p \in \mathbb{N}^+$
Let D be the duration of the loan in years, $D \in \mathbb{N}^+$
Lender's interest = 0.15 * p * D

It has been designed for the borrower's interest to be double the lender's; half the borrower's interest is given to the lender after a successful loan, and the other half is locked in the Hermes.sol contract, removing the tokens from the circulating supply which helps mitigate inflationary economics.

7.5.2 Collateral

The collateral the borrower has to put up is calculated from the price of the NFT and the lender's interest rate. Once the borrower has returned the NFT, they can only claim back tokens worth the price of the NFT.

Let p be the price of the NFT, $p \in \mathbb{N}^+$
Let i be the lender's interest, $i \in \mathbb{N}^+$
Collateral = p + i * 2

7.5.3 Liquidation

The lender of a loan can liquidate if the borrower has not returned the NFT by the end of the duration of the loan or the price of the NFT rises above the value of the collateral. This is to mitigate the lender's risk, so players are not discouraged from lending their NFTs.

Price Feed

If the price of the NFT increases to a value greater than the collateral, this puts the loaner at risk. Therefore, if the price of the NFT exceeds the collateral the lender is entitled to liquidate the borrower. The Market contract emits an event when an NFT is purchased, and a listener is used to listen for the emission of this event so the data can be collected for a price prediction algorithm to give price feeds to Hermes.sol.

Chainlink was considered to obtain off-chain data for the price of the NFTs but since LINK tokens are required to use this, a certain amount of liquidity would be required to be kept in the smart contracts to obtain LINK tokens. Instead, a Cron job is used to update the price of NFTs on loan every day. Although the price feed is not on-demand, using the Cron job reduces gas costs for players since the functions they call will not have to fetch off-chain data.

7.6 Game.sol

This contract will be the owner of multiple NFTs where one will be randomly selected to compete against players of the P2E. Originally, this contract would have used Chainlink to receive a random number to decide its NFT to play, then mint Drachma tokens if the player won the game, but a preferred method using the Pandora.sol contract is used to drastically reduce gas costs.

7.7 Pandora.sol

The game logic is handled completely off-chain to reduce the gas costs for the players. The player will choose one of their NFTs that they own to play against. Then a random NFT that the Game.sol contract owns is chosen to compete against the player. If the player wins the game, a hash is generated, using the keccak-256 hash function, and the deployer wallet private key generates a signature, which is used to verify this hash generated is genuine. Without this signature, anyone could generate a hash and withdraw an arbitrary number of Drachma tokens without having won a single game.

The address of the player and the signature are uploaded to a MongoDB collection. This is used so that when a player wants to withdraw their earnings, a request is made for this information and passed in as parameters to the Pandora contract. The

Pandora contract is used to claim earnings and validate the claim request. Only one state variable is updated in the Pandora contract, causing the gas consumption to be negligible compared to the alternative method that was considered using the Game contract.

7.8 Security & Gas Optimisations

7.8.1 Pull vs Push Payment

In Market.sol and Hermes.sol the design for the payment system has been carefully considered, as there are detrimental consequences if not implemented correctly. For example, external calls [61] may execute malicious code or fail deliberately, so every untrusted external call should be considered a potential security risk. To minimise this risk with payments, users should be made to withdraw their tokens instead of them being automatically sent.

The following code snippet is an example of a smart contract auction [61] implementing a push payment system that is vulnerable to Denial of Service (DoS) attacks. This is a common vulnerability found in many smart contracts and is only one reason of many as to why a pull payment system is favoured over a push payment system.

```
1  // THIS CONTRACT HAS AN INTENTIONAL VULNERABILITY, DO NOT COPY
2
3  contract Auction {
4    address public highestBidder;
5    uint256 public highestBid;
6    uint256 public endTime;
7
8    function bid() external payable {
9      require(block.timestamp < endTime, "[Auction]: Bidding has finished");
10     require(msg.value > highestBid, "[Auction]: Not a higher bid");
11     // XXX: IF THIS ALWAYS FAILS THEN NO ONE ELSE CAN BID
12     (bool sent, ) = highestBidder.call{ value: highestBid }("");
13     require(sent, "[Auction]: Transaction failed");
14     highestBidder = msg.sender;
15     highestBid = msg.value;
16   }
17 }
```

Figure 7.3: Auction with push payment system, adapted from [61]

The `bid()` function in the Auction contract 7.3 relies on the success of the transfer to the previous highest bidder, which can lead to malicious users taking advantage of this to ensure they win the auction.

```
contract Attacker {
  address public AUCTION_ADDRESS;
  Auction public auction = Auction(AUCTION_ADDRESS);

  function beginAttack() external payable {
    uint256 higherBid = auction.highestBid() + 10;
    auction.bid{ value: higherBid }();
  }

  fallback() external payable {
    revert("GUARANTEED TO WIN THE AUCTION NOW");
  }
}
```

Figure 7.4: DoS attack contract against Auction, inspired from [61]

Once the Attacker contract 7.4 calls the `beginAttack()` function, no other address is able to become the `highestBidder` because the Auction contract 7.3 requires the transaction to the previous `highestBidder` to succeed but the `fallback()` function will revert every time. Therefore, the Attacker contract is guaranteed to win the auction now. The following is a contract that overrides the Auction contract with an updated version of the `bid` function using a pull payment system which removes the DoS attack vulnerability.

```
contract AuctionPull is Auction {
  mapping(address => uint256) balances;

  function bid() override external payable {
    require(block.timestamp < endTime, "[Auction]: Bidding has finished");
    require(msg.value > highestBid, "[Auction]: Not a higher bid");
    balances[highestBidder] += highestBid;
    highestBidder = msg.sender;
    highestBid = msg.value;
  }

  // PULL PAYMENT
  function withdraw() external {
    uint256 amount = balances[msg.sender];
    balances[msg.sender] = 0;
    (bool sent, ) = msg.sender.call{ value: amount }("");
    require(sent, "[Auction]: Transaction failed");
  }
}
```

Figure 7.5: Auction with pull payment system, inspired from [61]

7.8.2 Zero Address

The special zero address [61, 67] is not owned by any user and is often used for token burning and a generic null address. Therefore, every function parameter of type `address` has been checked that it is not the special zero address with a `require` statement, to prevent any assets from being accidentally lost permanently. At the time of writing, the null address holds tokens of value more than $12 million.

7.8.3 Pragma Locking

Every solidity contract has a pragma at the top to specify the compiler version used. Floating pragmas [61] allow contracts to be deployed with a newer compiler version. This should be avoided due to the latest compiler having a higher risk of undiscovered bugs. Therefore, pragmas should be locked to the compiler version that the code has been tested with. For example, `pragma ^0.8.7` should be replaced by `pragma 0.8.7`.

7.8.4 tx.origin

In every contract `msg.sender` was used instead of `tx.origin` [61] as this can lead to failed attempts of authorisation in transactions. Furthermore, there is a possibility that `tx.origin` [68] will be removed from Ethereum, so it would not work with future releases.

7.8.5 Preferred Data Types

Due to the nature of the EVM [69], it is more gas efficient to use the data type `uint256` than others, such as `uint8`, because calculations by the EVM will convert number types to `uint256`. Therefore all unsigned integers have used type `uint256`.

7.8.6 External vs Public

The visibility keyword `external` was used instead of `public` wherever possible [61] due to the gas saved when operating with large arrays of data.

7.8.7 Slither API

In addition to the many security tools used on the smart contracts, custom scripts using the Slither API were created to detect additional exploits caused by bugs and validate rudimentary conditions. Wallet addresses used as input parameters were validated via built-in functions to ensure they targeted legal wallets, avoiding the possibility of losing funds permanently. Access control checks were carried out by inspecting function modifiers (which describe the conditions that allow the function to be called), such as `onlyOwner` and `onlyRole`, to ensure they were declared with

a specified signature including new asset generation through minting tokens. Additionally, smart contracts were scanned to verify that the necessary functions were implemented correctly against the ERC-20 and ERC-721 protocol standards.

7.9 Conclusion

The combination of the comprehensive security tools and processes described above ensures the smart contracts are designed to minimise the possibility of vulnerabilities throughout the entire application. The contracts also employ best practices for a stable market and economy of the reward tokens. This is additionally paired with intelligent optimisations to reduce gas costs for the player in every possible transaction.

Chapter 8

NFT Image Generation

The NFT images for Olympus are generated from layers. This is because automated designs reduce the burden on artists and ensure that a significant amount of content can be released, reducing future pressure on the development team. A single extra layer introduces a magnitude of additional combinations and can present a stream of fresh content for players at a low cost. Rarities can also be assigned for specific properties, such as a golden lightning bolt, enabling a more expansive and wide-ranging marketplace for the game.

The NFT images consist of three layers: background, shape, and trait, in addition to a random strength attribute in the range [0, 100]. The NFT images are generated by randomly choosing an image from each layer which are then combined. However, the NFT images must be unique, so to check for this, the name stored in the metadata is the concatenation of the file names used in each layer.

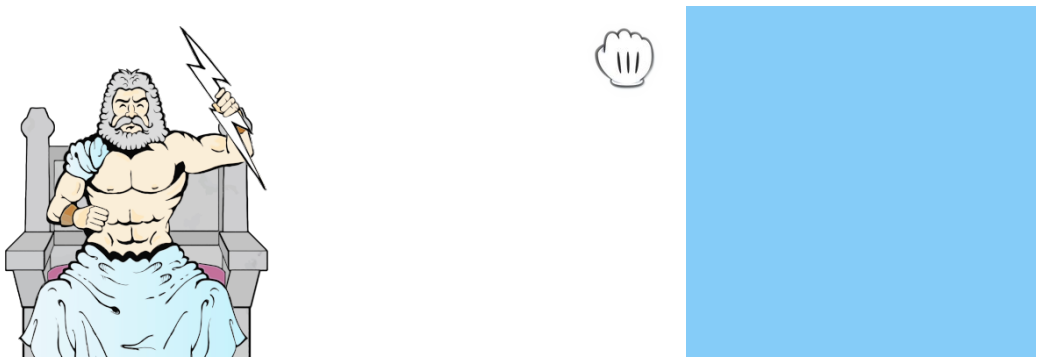

Figure 8.1: Layers used to generate NFT images

Due to the time and space complexity of the generation of the images, the server would be put under great stress to generate all combinations of layers. Therefore, to reduce the load on the server, images are generated in batches of size 100. To allow the images to be batch-generated, a listener is used to listen for when a token is minted (the transfer event from the zero address) and every 95^{th} time another batch is generated.

Before a batch generation, the metadata of all existing NFTs are fetched and the names are added to a hash map, so new images can check if they are unique in constant time. Images are generated until 100 unique images are chosen.

MongoDB [70] is an open-sourced, highly scaleable document database with a flexible schema approach, removing the need to configure a database. Alternative to storing data in tables of rows or columns, each record is described as a BSON, which can be retrieved in a JSON format. Due to MongoDB being able to accommodate high data loads, it makes it the perfect choice for storing the metadata of each image for the P2E.

Amazon S3 is a centralised cloud object storage platform. A decentralised alternative could be used so the P2E is fully decentralised, but the scalability, performance and security of S3 outweigh the benefit of decentralisation. Also, S3 has a simplified process to upload images and automatically generates the URL once uploaded. Thus, S3 was chosen.

Chapter 9

Price Prediction

An algorithm was created to predict the selling price of the NFTs. This will be used to give players information on what foreseeable price their NFT may sell for, and allow lenders to liquidate borrowers.

Initially, a static price prediction algorithm was used. However, this static prediction is not suitable to be used for lenders to liquidate borrowers on the DeFi platform, as the NFT price will not adapt to the ever-changing market conditions that it is in. Thus, a dynamic model was chosen to replace this using the time of the sell, past selling price, trait and strength of the NFT as features to predict the future selling price.

9.1 Synthetic Data

There is no available data on the selling prices of the Olympus NFT collection to train my models, so synthetic data was created for the price prediction model. The selling time was calculated by subtracting a random integer in the range of one to two weeks from the current time. For the trait, a simple random integer generator in the range [0,2] was used with each number corresponding to rock, paper, and scissors respectively. Similarly for strength in the range [0,100].

9.1.1 Previous Selling Price

Initially, to generate synthetic data for the prices, a uniform distribution in the range [0, 500], was used. However, this does not represent NFT selling prices accurately because typically an NFT would sell around its floor price or considerably higher due to spontaneous reasons.

After the issues with a uniform distribution were noticed, a Gaussian distribution was used with a mean of 300 (the floor price of the collection in Drachma tokens) and a standard deviation of 100 to generate the synthetic price data instead.

After experimenting with the synthetic data generation, it was noticed that the data did not replicate price trends over a time period.

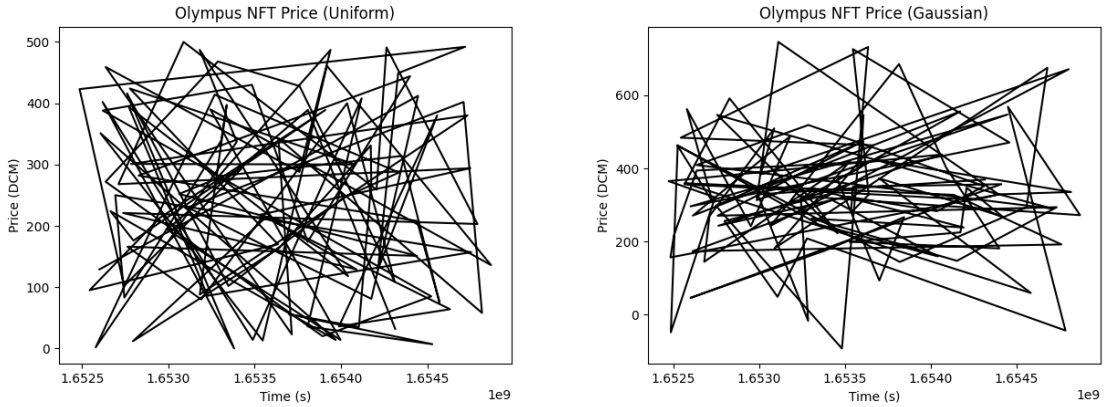

Figure 9.1: Synthetic data created using uniform and Gaussian distributions

A new algorithm was developed which generates multiple selling prices within a 15% threshold of a base price, then randomly increases or decreases the base price by 20% to repeat the process for the "previous day". This process is repeated to generate realistic rises and dips in price for five months.

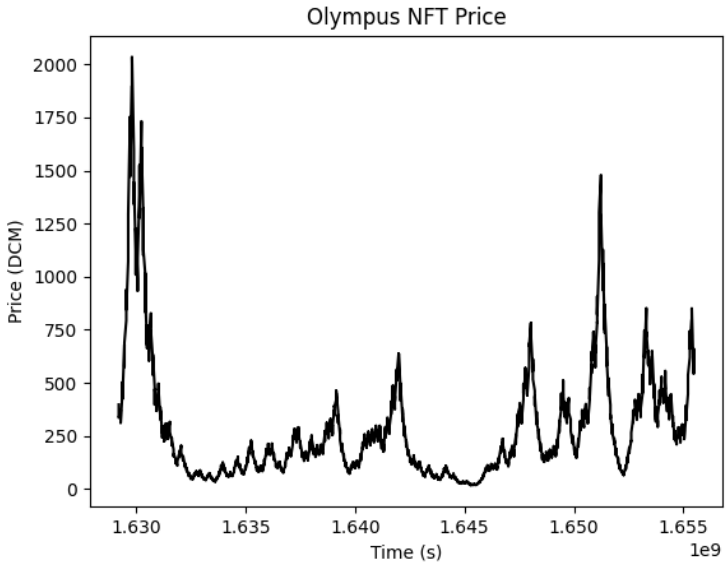

Figure 9.2: Synthetic data created with realistic price trend

To aid with debugging, a seed (of value one) from the random library was selected to ensure the results are reproducible.

9.2 Models

For the dynamic price prediction model, three unique models were chosen to experiment with to compare and choose the best performing one. These were a linear, decision tree and a neural network model. The linear and decision tree models used were models from the scikit learn [71] module in Python.

To achieve better results, a neural network was created with custom hyperparameters, using the Keras [72] library. Moreover, a random search was conducted to find the optimal values for the number of epochs, the learning rate and the batch size. The information for the hyperparameter tuning is displayed in table 9.3 below.

Hyperparameter	Values Tested	Optimal Value
# epochs	[32, 64, 256]	32
learning rate	10^{-i}, $\forall$ i $\in$ [2,6)	0.01
batch size	2^i, $\forall$ i $\in$ [3,7)	64

Table 9.1: Hyperparameters chosen for the neural network

Table 9.2 shows the evaluation of the models using the MSE and R^2 values.

Model	MSE	R^2
Linear	83216.17	0.00
Decision Tree	11568.26	0.86
Neural Network	101551.55	0.00

Table 9.2: Evaluation of models

During experiments with the previous models, a more suitable model for price prediction was found: LSTM. For this model, a random search was conducted for hyperparameter tuning, shown below in 9.3, and early stopping was used to prevent overfitting. After experimenting with the architectures of four unique networks, the LSTM model was chosen for the price prediction.

Hyperparameter	Values Tested	Optimal Value
# epochs	[32, 64, 256]	64
learning rate	10^{-i}, $\forall$ i $\in$ [2,6)	0.01
batch size	2^i, $\forall$ i $\in$ [3,7)	32
activation function	[tanh, sigmoid]	sigmoid
recurrent activation function	[tanh, sigmoid]	tanh

Table 9.3: Hyperparameters chosen for the neural network

Chapter 10

Code Development, Testing & Deployment

10.1 Code Development

This project's intention of showing optimised code for developers to reduce gas costs and maintain an exceptional standard of security required a strong focus on high-quality code. The strive for consistency across the code base combined with adhering to industry standards led to constructing efficient pipelines for checking formatting, linting, security and common code conventions.

Prettier [73] and autopep8 [74] were used with custom rules to format all relevant files to strict standards. Additionally, ESlint [75], Pylint [76] and Solhint [77] were used to provide static code analysis to identify bugs, common security concerns and bad practices. These were augmented with custom configurations to cover a broader scope of the code base including frontend components and testing frameworks.

A coherent understanding of the code for readers was another essential aspect of code development. This was achieved through following commenting conventions for all languages including the Solidity NatSpec [78] commenting format and Google-style [79] Python Docstrings. Linting rules allowed automated checks on comments and ensured that they were present in the required code sections. Suggestions from the linters also ironed out minor errors in comments such as misspelt parameter names.

Custom Git hook scripts [80] were implemented from scratch to automate the process of formatting and linting using client-side hooks. Using the pre-commit hook, the code is automatically formatted and if linting fails then the pre-commit hook will exit with failure, preventing the code from being committed. By using this automation, a higher standard of code quality was enforced. Moreover, the commit-msg hook was used to enforce a commit message convention, drastically improving clarity and readability by making it easier to identify the type of change that was made.

10.2 Testing

Correct code functionality was of the utmost importance for this project, as it interacts with assets that have real-world value. Comprehensive automated testing was employed at all levels of the code to ensure that it behaved as intended. This led to using a collection of tools which allowed for fine-grained unit testing and broader integration tests.

Hardhat [81] was utilised to set up a powerful Ethereum development environment reflecting how the code will run when deployed on the mainnet. The Ethers package, from the Hardhat library, allowed for a wide range of smart contract interactions to be tested. Mocha [82] was then used for coherent and exhaustive automated tests for all smart contracts. This was paired with EVM time manipulation to expand the types of tests that can be carried out to include functions that rely on `block.timestamp`. Some tests included checking if the reverts were successful, which closed opportunities for attackers to exploit the smart contract's integrity. Finally, access control was also tested thoroughly to uphold security.

Jest [83] was also used to test the server-side logic effectively. The code coverage tool ensured that there were no gaps in the testing and led to a more robust system. The mocking functions improved the brevity of the code and the clear context on test failures sped up development time. Overall, this improved the code quality and meant that the application was tested rigorously to prevent bugs.

10.3 CI/CD

Throughout the project, GitHub actions [84] was used as the CI/CD platform to automate a significant number of checks, tests and the deployment of the application. Custom actions were written to wire different stages of the pipeline together efficiently. The pipeline was also parallelised and split to reflect the structure of the code base, making it easier to pinpoint particular issues. Moreover, the deployment was reserved for the master branch meaning that only pure working code could be released.

For every push, compilation was checked for the smart contracts, client-side code and server-side code. The code base was also scanned to identify any formatting and linting issues based on the custom configurations that were previously mentioned. All tests were run to verify that a change did not break any functionality. This was paired with the Echidna GitHub action [85] to test whether the application maintained expected invariants for a large number of inputs. Additionally, the pipeline ran a stream of static analysis tools using the Slither GitHub action [86] and purpose-built Slither scripts to uncover security vulnerabilities.

The CI/CD was an integral part of making a successful application and this project strongly recommends to blockchain developers to build comprehensive pipelines to automate checks and tests. This speeds up development time by revealing possible bugs or security vulnerabilities when code is pushed. The result is a clean, correct and secure application.

10.4 Conclusion

The combination of strict automated testing, formatting, linting and vulnerability scans has led to a cleaner and safer codebase. The overall application benefits from optimised transactions to save its users from paying high gas costs. It also protects its users by safeguarding against vulnerabilities which attackers can exploit to gain access and possibly destroy the game economy. Automation played a key part in the succinct development process and helped reduce human error from being a factor to introducing vulnerabilities or incorrect functionality.

Chapter 11

Evaluation

11.1 Risks

- Smart contracts are immutable once deployed, so any vulnerability missed is a severe security issue to the Olympus P2E. As demonstrated later in 11.2.2, the smart contracts developed have been thoroughly assessed using a well-renowned static analysis tool.
- Certain aspects of the project are centralised, which, on the one hand, can repulse people in the crypto community due to their appreciation of the decentralised technology. On the hand, this was implemented in order to reduce gas consumption per transaction, resulting in cheaper fees for the Olympus player base
- In the current implementation, it is up to the borrower to initiate the return process for the borrowed NFT asset to give it back to its rightful owner. This could result in a scenario whereby a malicious borrower deviates from honest behaviour and neglects to return the asset to the owner. Although this behaviour is unlikely to occur, let alone impact the lender because the loan is already over collateralised

11.1.1 External Risks

- The Ethereum network is notorious for high gas fees and can easily become congested, causing the game to be costly to play
- AWS S3 and Elastic Beanstalk are centralised cloud services, so they are single points of failure
- MongoDB is a centralised database, so it is vulnerable to the same issue as the AWS services
- Multiple smart contracts utilise the OpenZeppelin library so any vulnerability would negatively impact the P2E
- All smart contracts are deployed on the EVM, so any fault or exploit with the EVM would adversely affect the P2E

11.2 Smart Contracts

11.2.1 Testing & Coverage

The following table 11.1 displays the test coverage information for the smart contracts. This is an informative metric for smart contract testing because untested functionality could lead to undesired behaviour and malicious users exploiting this.

Contract Name	Statements	Branch	Functions	Lines
Drachma	100	50	100	100
Olympus	91	83	80	91
Hermes	95	75	91	95
Market	97	88	90	97
Game	100	50	100	100
Pandora	100	88	100	100
Total	97	72	94	97

Table 11.1: Test coverage summarised, generated using [87]

11.2.2 Security

As mentioned previously, the security of smart contracts is one of the most significant factors in blockchain development. Hence, the security evaluation is by far the most significant. The Slither tool reported warnings in the contracts used from the OpenZeppelin library and dangerous `block.timestamp` comparisons in Hermes.sol. The `block.timestamp` can be safely ignored for previously mentioned reasons. Similarly, the OpenZeppelin contracts are open-source and they have been audited multiple times by the community, so it is safe to ignore these as well.

Comparing the top three P2E projects by market cap [88] as well as projects previously mentioned in the background, a group of five additional projects are used to compare to Olympus. Of the five, Upland had no audit available and also had no public code to potentially check independently, so it is omitted from the table 11.2. A weighting is applied according to the category of the issue found, starting with informational as *one* and enumerating until critical at *five*. Of the projects that had a security report available, Olympus ranks 2^{nd} lowest in weighting.

Project	Olympus	Axie Infinity	Alien Worlds	Decentraland	Sandbox
Critical	0	0	0	0	0
Major	0	0	0	2	1
Medium	0	0	0	0	1
Minor	0	0	0	0	5
Informational	4	5	0	2	13
Total	4	5	0	4	20
Weighted Total	4	5	0	10	30

Table 11.2: Security report comparison [89, 90, 91, 92]

A magnitude of custom invariants was written in order to carry out fuzzing tests. The Echidna tool attempted to break these with random transactions but it was discovered that all invariants held for the smart contract tested.

Contract Name	Tests Passing
Olympus	100000 / 100000
Hermes	550000 / 550000
Market	200000 / 200000
Pandora	100000 / 100000

Table 11.3: Echidna fuzzing report

11.2.3 Gas Consumption

Reducing gas consumption was considered throughout the whole development of Olympus and was a core focus, as players do not want to pay extreme gas fees to interact with the P2E game.

The hardhat-gas-reporter [93] tool is used to report gas used in each test run. A summary of the average gas usage per contract can be found in the table 11.4 below.

Contract Name	# Function Calls	Average Gas (Gwei)
Drachma	20	156223
Olympus	17	361557
Hermes	19	748527
Market	12	506582
Pandora	7	136932

Table 11.4: Summarised gas report using [93]

The hardhat-gas-reporter [93] tool and information obtained from [94] was used to compare the gas consumption between Axie Infinity and Olympus. The results below indicate that in 4/7 of the different transactions, Olympus outperforms Axie

Infinity, a market leader in its field. The "Add NFT Listing" transaction, in particular, demonstrates the largest gap.

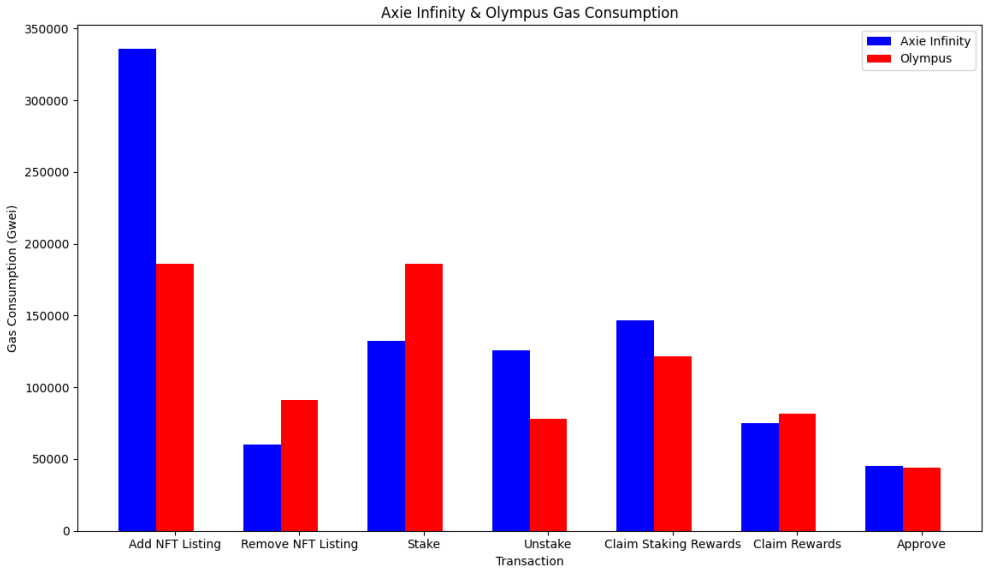

Figure 11.1: Gas Consumption comparison between Axie Infinity and Olympus

11.3 NFT Image Generation

Players should have access to their images instantly so they are able to play whenever they please. If the image generation is sluggish, then players who purchase an NFT may not have the associated image available, causing a distaste for the P2E. Therefore, ensuring the batch process is completed as quickly as possible is critical.

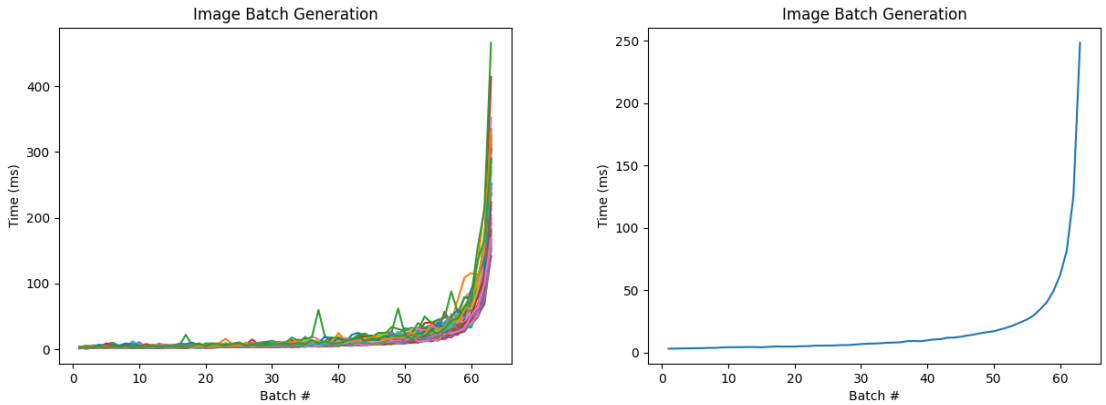

Figure 11.2: Batch Generation Time & Average Batch Generation Time

11.4 Prediction Model

The LSTM model was evaluated using MSE and R^2, the same as previous models experimented with. In comparison to these, the LSTM model clearly outperforms them in both metrics used, as shown in 11.5.

Model	MSE	R^2
Linear	83216.17	0.00
Decision Tree	11568.26	0.86
Neural Network	101551.55	0.00
LSTM	91.19	0.99

Table 11.5: Evaluation of models

The graph below 11.3 shows that the model's predicted NFT prices match the actual prices almost perfectly, achieving very low losses relative to the other models described in this report.

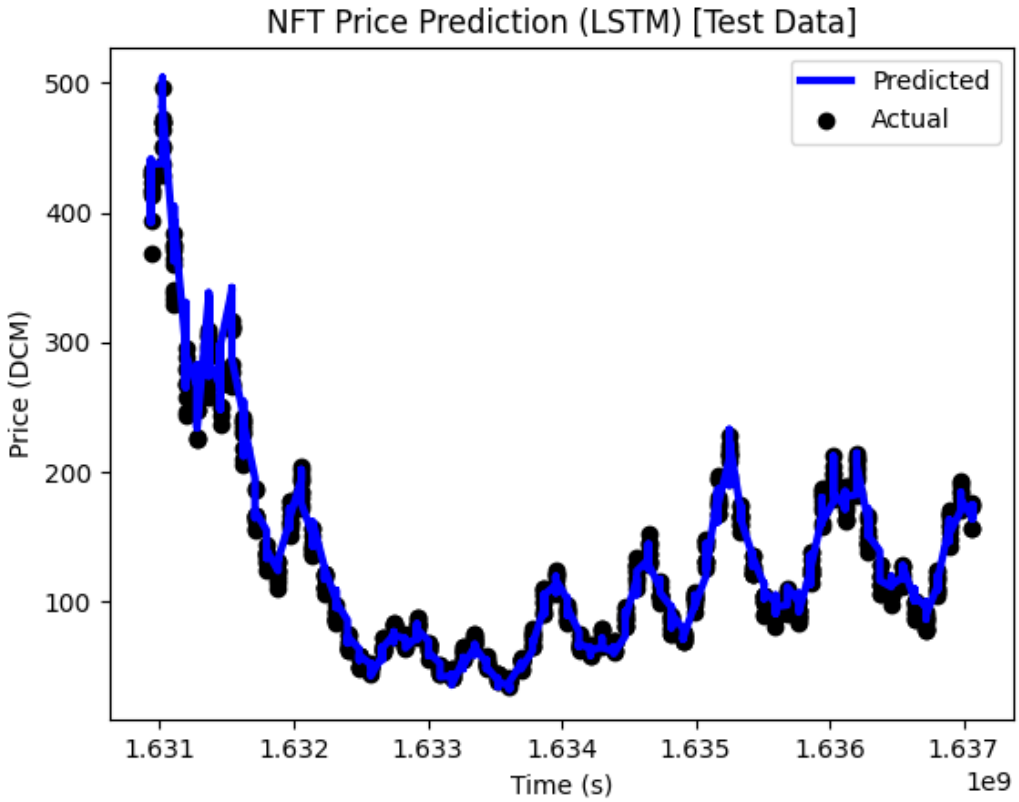

Figure 11.3: LSTM Plot

11.5 User Feedback

A core part of game development is user feedback. This is paramount for game developers as they can gain useful information to better understand their player base. A questionnaire was used to gauge responses on the Olympus P2E game, and based on their feedback, several features and quality-of-life improvements were implemented.

Figure 11.4: Key feedback for Olympus

From the figure above, it can be seen that the majority of the players asked were against moving game logic on-chain due to the increase in gas costs it would cause. However, the percentage of people who answered yes increased for wanting a decentralised image hosting platform to be used.

Chapter 12

Conclusion

The project has successfully amalgamated best practices to be used as an exemplar for future developers of vulnerability-averse cryptocurrency P2E games. A major concern with cryptocurrency projects is developing secure smart contracts. This project addresses the issue with a combination of industry-leading techniques including static analysis, fuzzing, comprehensive testing, and carefully constructed code designs such as using the pull payment system. It also introduces scripts which ensure that deployed code is checked thoroughly for dangerous vulnerabilities in Solidity development. It is hoped that the range of security procedures employed in this P2E is used as inspiration for future projects and encourages developers to take security seriously in blockchain-based games.

Another issue that is widespread in Ethereum-based applications, is the potential for gas prices to exceed the reward payout for a player. This can cause a winning player to lose most or all of their earnings if not implemented with care. This project ensured that the solidity contracts underwent careful analysis to introduce code optimisations, effectively reducing the computational cost of running them hence reducing the gas consumption. Common optimisations include [69] replacing arrays with mappings, packing variables, off-chain calculations and preferred data type usage such as `uint256`. Another key takeaway for future blockchain-based game developers from this project is using these optimisations to ensure their code minimises gas consumption and, as a consequence, reduces the impact of their games on the environment.

As P2E blockchain games require an asset, such as an NFT to play, players require an understanding of the potential value of these assets. An issue that arises in current iterations of these games are the turbulent market conditions affecting the price of the assets, and clouding players' understanding of its value. This project attempts to solve this issue with a dedicated price prediction model for the game's NFTs, using LSTM machine learning. It is encouraged that fellow developers use this as an example to think more carefully about similar methods in deobfuscating the values of rewards and playable assets. This complexity reduction will attempt to break down the barrier to entry for gamers and enable the industry to grow further.

12.1 Future Plans

The P2E ecosystem is in its early stages and has a vast majority of features still unexplored. Many of these can be considered for future work if development was to continue on the Olympus P2E.

A smart contract audit [95] will typically consist of the code being examined, and a report produced to the creators to fix any issues found. After, a final report is released with information regarding any outstanding errors and the work that was done to adjust any security or performance issues. Since all transactions of the blockchain are immutable and funds cannot be retrieved if they have been stolen, the Olympus project would benefit greatly from a smart contract audit from a respectable company such as ConsenSys Diligence or OpenZeppelin.

For development purposes, the Olympus project was deployed on the Ethereum Rinkeby test network. However, a future consideration could be to promote the P2E to gain popularity and then deploy it on Ethereum's main network for people to play with. Before deploying on the main network, many more features could be introduced, such as higher-priced loot boxes for a greater chance to obtain a rare NFT or introducing a player vs player game mode.

Chapter 13

Appendix A

13.1 MagicNumber.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

/// @title Magic Number Game
/// @author Alex Richardson
contract MagicNumber {
  address private _owner;
  uint256 public deadline;
  uint256 private _target;
  uint256 private _winningPool;
  uint256 public total;
  bool private _guessed;
  mapping(address => uint256) public stakes;
  mapping(address => bool) public correct;

  event Create(uint256 number, uint256 deadline, address owner);
  event Bet(uint256 guess);

  constructor(
    uint256 number,
    uint256 endtime,
    address owner
  ) payable {
    require(msg.value > 0, "[MagicNumber] Invalid price.");
    require(owner != address(0), "[MagicNumber] Invalid addres.");
    _owner = owner;
    deadline = endtime;
    _target = number;
    _winningPool = 0;
    total = msg.value;
```

```
31      stakes[owner] = msg.value;
32      emit Create(number, endtime, owner);
33    }
34
35    /// @notice Place a bet on a number
36    /// @param guess The number that has been guessed
37    /// @dev Send ETH to the contract
38    function bet(uint256 guess) external payable {
39      require(block.timestamp <= deadline, "[MagicNumber] Game has finished.");
40      require(msg.value > 0, "[MagicNumber] Invalid price.");
41      require(0 < guess && guess <= 10, "[MagicNumber] Invalid guess.");
42      require(stakes[msg.sender] == 0, "[MagicNumber] You have already bet.");
43      stakes[msg.sender] = msg.value;
44      total += msg.value;
45      if (guess == _target) {
46        _guessed = true;
47        _winningPool += msg.value;
48        correct[msg.sender] = true;
49      }
50      emit Bet(guess);
51    }
52
53    /// @notice Claim for the owner of the game
54    /// @dev Should only be called in `claim`
55    function _claim() internal {
56      if (!_guessed) {
57        uint256 winnings = total;
58        total = 0;
59        (bool sent, ) = msg.sender.call{ value: winnings }("");
60        require(sent, "[MagicNumber]: Failed ETH transaction");
61      }
62    }
63
64    /// @notice Claim the reward for guessing the correct number
65    /// @dev _winningPool is split between all participants that
66    ///      guessed correctly
67    function claim() external {
68      require(
69        block.timestamp >= deadline,
70        "[MagicNumber] Game has not finished."
71      );
72      require(stakes[msg.sender] > 0, "[MagicNumber] You have not placed a bet.");
73      if (msg.sender == _owner) {
74        _claim();
75      } else if (correct[msg.sender]) {
```

```
76        uint256 reward = (stakes[msg.sender] * total) / _winningPool;
77        correct[msg.sender] = false;
78        stakes[msg.sender] = 0;
79        (bool sent, ) = msg.sender.call{ value: reward }("");
80        require(sent, "[MagicNumber]: Failed ETH transaction");
81      }
82    }
83  }
```

13.1.1 MagicNumber.test.ts

```
1  import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
2  import { fail } from "assert";
3  import { expect } from "chai";
4  import { ethers } from "hardhat";
5  import { MagicNumber } from "../typechain-types";
6
7  function checkSigner(signer: SignerWithAddress | undefined):
   ↪  SignerWithAddress {
8    if (!signer) {
9      fail("Signer is undefined");
10   }
11   return signer;
12 }
13
14 const TARGET = 5;
15 const INVALID_GUESS = 11;
16 const INCORRECT_GUESS = 1;
17 const BET_DURATION = 3600; // seconds
18 const MS_TO_SECS = 1000;
19 const END_TIME = Math.floor(Date.now() / MS_TO_SECS) + BET_DURATION;
20
21 describe("MagicNumber Contract", function () {
22   let magicNumberContract: MagicNumber;
23   let creator: SignerWithAddress;
24   let betterOne: SignerWithAddress;
25   let betterTwo: SignerWithAddress;
26   let noBetter: SignerWithAddress;
27
28   before("Deploy contract", async function () {
29     const MagicNumberContract = await
       ↪  ethers.getContractFactory("MagicNumber");
30     magicNumberContract = await MagicNumberContract.deploy(
31       TARGET,
32       END_TIME,
```

```
33        "0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266",
34        {
35          value: ethers.utils.parseEther("1"),
36        }
37      );
38      await magicNumberContract.deployed();
39      const [signerOne, signerTwo, signerThree, signerFour] = await
        ↪  ethers.getSigners();
40      creator = checkSigner(signerOne);
41      betterOne = checkSigner(signerTwo);
42      betterTwo = checkSigner(signerThree);
43      noBetter = checkSigner(signerFour);
44    });
45
46    describe("Bet", function () {
47      it("Should not allow addresses to bet for free", function () {
48        expect(
49          magicNumberContract.connect(betterOne).bet(INCORRECT_GUESS, {
50            value: ethers.utils.parseEther("0"),
51          })
52        ).to.be.revertedWith("[MagicNumber] Invalid price.");
53      });
54
55      it("Should not allow guesses outside of (0,10]", function () {
56        expect(
57          magicNumberContract.connect(betterOne ||
          ↪  "").bet(INVALID_GUESS, {
58            value: ethers.utils.parseEther("1"),
59          })
60        ).to.be.revertedWith("[MagicNumber] Invalid guess.");
61      });
62
63      it("Successfully betting should increase the contract's balance",
      ↪  async function () {
64        const balanceBefore = await magicNumberContract.total();
65        const BET_MONEY = ethers.utils.parseEther("1");
66        await magicNumberContract.connect(betterOne ||
          ↪  "").bet(INCORRECT_GUESS, {
67          value: BET_MONEY,
68        });
69        expect(await magicNumberContract.total()).to.equal(
70          balanceBefore.add(BET_MONEY)
71        );
72      });
73
```

```
74      it("Should not allow the same address to bet twice", async
        ↪  function () {
75        await magicNumberContract.connect(betterTwo ||
          ↪  "").bet(INCORRECT_GUESS, {
76          value: ethers.utils.parseEther("1"),
77        });
78        expect(
79          magicNumberContract.connect(betterTwo ||
            ↪  "").bet(INCORRECT_GUESS, {
80            value: ethers.utils.parseEther("1"),
81          })
82        ).to.be.revertedWith("[MagicNumber] You have already bet.");
83      });
84
85      it("Should not allow addresses to bet after the deadline", async
        ↪  function () {
86        await ethers.provider.send("evm_increaseTime", [BET_DURATION]);
87        await ethers.provider.send("evm_mine", []);
88        expect(
89          magicNumberContract.connect(betterOne ||
            ↪  "").bet(INCORRECT_GUESS, {
90            value: ethers.utils.parseEther("1"),
91          })
92        ).to.be.revertedWith("[MagicNumber] Game has finished.");
93      });
94    });
95
96    describe("Claim", function () {
97      it("Should not allow an address to claim without having made a
        ↪  bet", function () {
98        expect(
99          magicNumberContract.connect(noBetter || "").claim()
100       ).to.be.revertedWith("[MagicNumber] You have not placed a
          ↪  bet.");
101     });
102
103     it("Should send all staked money to creator if no correct
        ↪  guesses", async function () {
104       await magicNumberContract.claim();
105       expect(await magicNumberContract.total()).to.equal(0);
106     });
107
108     it("Should not allow creator to claim twice", async function () {
109       const balanceBefore = await creator.getBalance();
110       await magicNumberContract.claim();
```

```
111       const balanceAfter = await creator.getBalance();
112       expect(balanceAfter.toBigInt() <=
          ↪  balanceBefore.toBigInt()).to.be.true;
113     });
114   });
115 });
```

13.1.2 MagicNumber.sol Echidna Tests

```
1  function echidna_check_owner_is_not_zero() public view returns (bool) {
2    return _owner != address(0);
3  }
4
5  function echidna_check_deadline_is_positive() public view returns (bool) {
6    return deadline > 0;
7  }
8
9  function echidna_check_target_is_between_one_and_ten()
10   public
11   view
12   returns (bool)
13 {
14   return 0 < _target && _target <= 10;
15 }
16
17 function echidna_check_winningPool_is_non_negative()
18   public
19   view
20   returns (bool)
21 {
22   return 0 <= _winningPool;
23 }
24
25 function echidna_check_winningPool_is_lte_total()
26   public
27   view
28   returns (bool)
29 {
30   return _winningPool <= total;
31 }
32
33 function echidna_check_total_is_non_negative() public view returns (bool) {
34   return 0 <= total;
35 }
36
```

```
37 function echidna_check_if_guessed_total_is_positive()
38   public
39   view
40   returns (bool)
41 {
42   if (_guessed) {
43     return total > 0;
44   }
45   return true;
46 }
47
48 function echidna_check_stakes_is_non_negative() public view returns (bool) {
49   return stakes[msg.sender] >= 0;
50 }
51
52 function echidna_check_stakes_lte_total() public view returns (bool) {
53   return stakes[msg.sender] <= total;
54 }
55
56 function echidna_check_if_correct_stakes_is_positive()
57   public
58   view
59   returns (bool)
60 {
61   if (correct[msg.sender]) {
62     return stakes[msg.sender] > 0;
63   }
64   return true;
65 }
```

Chapter 14

Appendix B

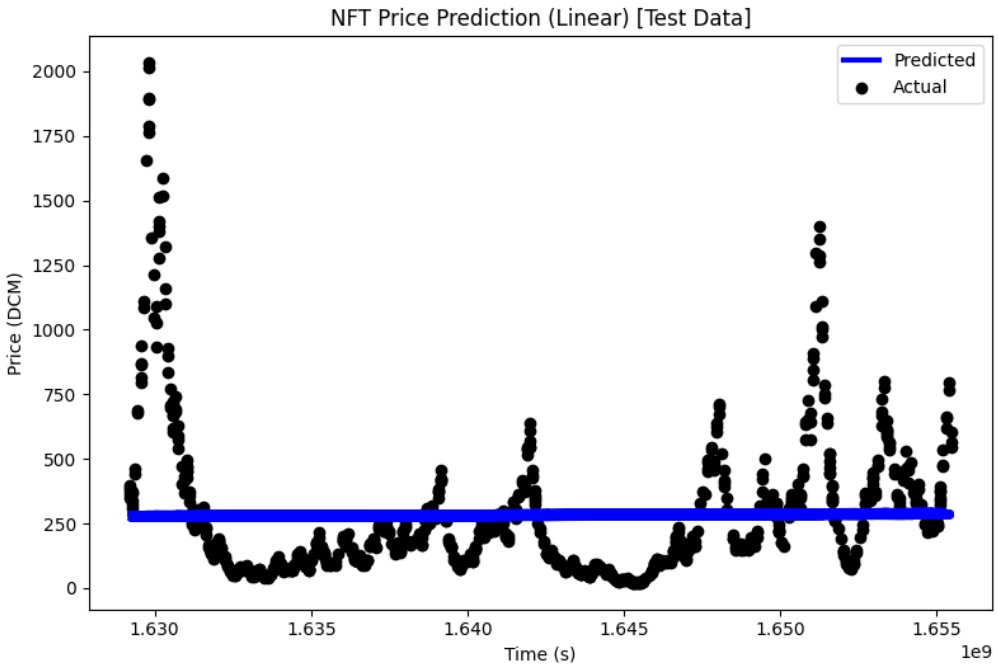

Figure 14.1: Linear plot

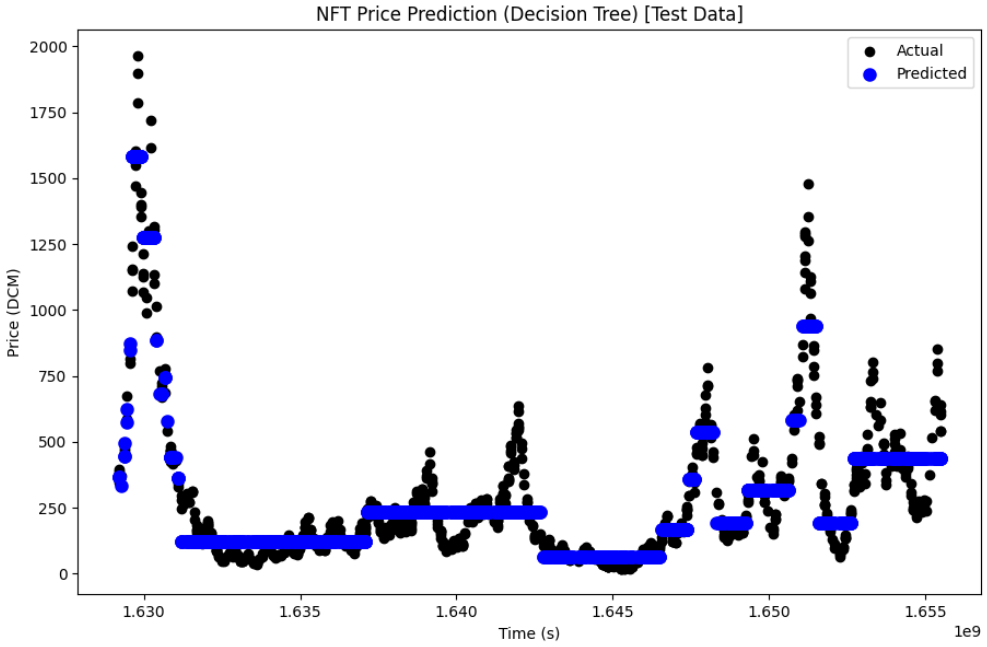

Figure 14.2: Decision Tree plot

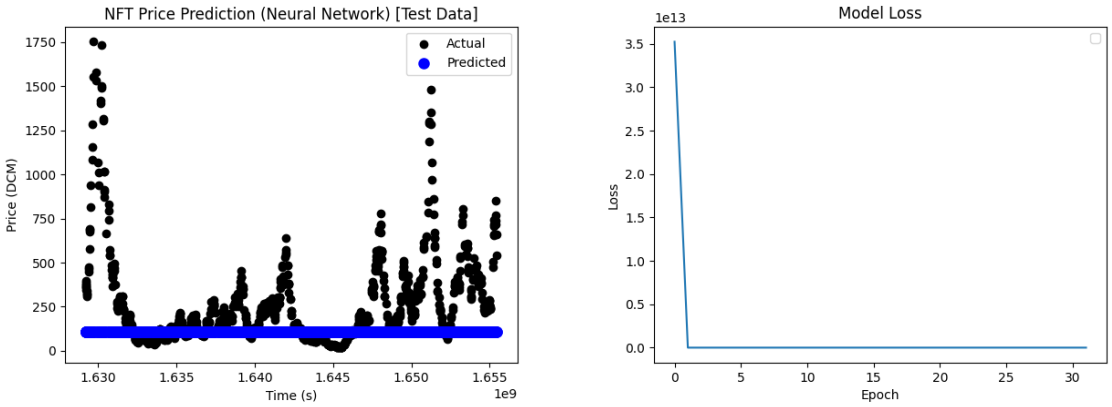

Figure 14.3: ANN plot

Bibliography

[1] Vitalik Buterin. Ethereum Whitepaper, 2013. URL https://ethereum.org/en/whitepaper/. pages 1, 4

[2] Cryptomeda. What is play-to-earn? the medawars game explained, December 2021. URL https://cryptomedatech.medium.com/what-is-play-to-earn-the-medawars-game-explained-2618787a06bd. pages 1

[3] Doug Petkanics. The benefits of decentralization, November 2016. URL https://petkanics.medium.com/the-benefits-of-decentralization-88a0b5d0fd39. pages 1

[4] Prashant Jha. The aftermath of Axie Infinity's $650M Ronin Bridge hack, April 2022. URL https://cointelegraph.com/news/the-aftermath-of-axie-infinity-s-650m-ronin-bridge-hack. pages 2

[5] Justin Vallejo. Bitcoin in crosshairs as Democrats target crypto mining in green energy push, January 2022. URL https://www.independent.co.uk/news/world/americas/bitcoin-crypto-firms-democrats-b1997453.html. pages 3

[6] Paul Wackerow. Intro to Ethereum, December 2021. URL https://ethereum.org/en/developers/docs/intro-to-ethereum/. pages 4

[7] Paul Wackerow. Proof-of-work (PoW), December 2021. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/. pages 5

[8] Paul Wackerow. Mining, December 2021. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/mining/. pages 5

[9] Joshua. Proof-of-stake (Pos), January 2022. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/. pages 5, 6

[10] Paul Wackerow. Gas and fees, December 2021. URL https://ethereum.org/en/developers/docs/gas/. pages 6

[11] Paul Wackerow. Ethereum Virtual Machine (EVM), December 2021. URL https://ethereum.org/en/developers/docs/evm/. pages 7

[12] Solidity Team. Solidity programming language, 2022. URL https://soliditylang.org//. pages 7

[13] Ethereum. Solidity — Solidity 0.8.15 documentation, March 2022. URL https://docs.soliditylang.org/en/latest/. pages 7

[14] Paul Wackerow. Introduction to smart contracts, December 2021. URL https://ethereum.org/en/developers/docs/smart-contracts/. pages 7

[15] Paul Wackerow. Smart contract security, December 2021. URL https://ethereum.org/en/developers/docs/smart-contracts/security/. pages 7, 8, 9, 10

[16] Alex Hern. '$300m in cryptocurrency' accidentally lost forever due to bug. *The Guardian*, November 2017. ISSN 0261-3077. URL https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether. pages 7

[17] Wolfie Zhao. $30 million: ether reported stolen due to parity wallet breach, July 2017. URL https://www.coindesk.com/markets/2017/07/19/30-million-ether-reported-stolen-due-to-parity-wallet-breach/. pages 7

[18] Paul Wackerow. A guide to smart contract security tools, December 2021. URL https://ethereum.org/en/developers/tutorials/guide-to-smart-contract-security-tools/. pages 10

[19] Sharif Elfouly. Mythril readme, April 2022. URL https://github.com/ConsenSys/mythril/blob/develop/README.md. pages 10

[20] OpenZeppelin. Contracts - OpenZeppelin Docs, 2021. URL https://docs.openzeppelin.com/contracts/4.x/. pages 10, 11, 12, 13, 30, 31

[21] Paul Wackerow. ERC-20 token standard, December 2021. URL https://ethereum.org/en/developers/docs/standards/tokens/erc-20/. pages 10

[22] Joshua. ERC-721 non-fungible token standard, December 2021. URL https://ethereum.org/en/developers/docs/standards/tokens/erc-721/. pages 11

[23] Azuki. Introducing ERC721A: an improved ERC721 implementation, 2022. URL https://www.azuki.com/erc721a. pages 12

[24] Septima Elle. NFT rug pull: what it is and how to avoid it, May 2022. URL https://medium.com/orderinbox/nft-rug-pull-what-it-is-and-how-to-avoid-it-607b787ca5e3. pages 12

[25] Ola. What is ERC-721R? About the safest new NFT standard, April 2022. URL https://nftevening.com/what-is-erc-721r-about-the-safest-new-nft-standard/. pages 12

[26] Exo Digital Labs . ERC721R: bringing greater accountability to NFT creators, April 2022. URL https://erc721r.org/. pages 12

[27] What are NFTs and why are some worth millions? *BBC News*, September 2021. URL https://www.bbc.com/news/technology-56371912. pages 13

[28] Jack Dorsey: Bids reach $2.5m for Twitter co-founder's first post. *BBC News*, March 2021. URL https://www.bbc.com/news/world-us-canada-56307153. pages 13

[29] Victor Kao. How NFTs can disrupt gaming, June 2021. URL https://builtin.com/blockchain/how-nfts-can-disrupt-gaming. pages 13

[30] Coinbase. DAOs: Social networks that can rewire the world, December 2021. URL https://blog.coinbase.com/daos-social-networks-that-can-rewire-the-world-128b73732547. pages 14

[31] Ethereum. Decentralized Autonomous Organizations (DAOs), January 2022. URL https://ethereum.org/en/dao/. pages 14

[32] Coinbase. What is DeFi?, 2022. URL https://www.coinbase.com/learn/crypto-basics/what-is-defi. pages 14

[33] Ethereum. Decentralized finance (DeFi), May 2022. URL https://ethereum.org/en/defi/. pages 14

[34] Chainlink. Hybrid smart contracts explained, May 2021. URL https://blog.chain.link/hybrid-smart-contracts-explained/. pages 15

[35] Solana. Solana - scalable blockchain infrastructure, 2022. URL https://solana.com/. pages 15

[36] Chase Barker. Getting started with Solana development, August 2021. URL https://solana.com/news/getting-started-with-solana-development. pages 15

[37] Cardano. Cardano docs, 2022. URL https://docs.cardano.org/. pages 15

[38] Axie Infinity. Axie community alpha: getting started!, January 2020. URL https://medium.com/axie-infinity/axie-community-alpha-getting-started-26aa9c6cbd43. pages 16

[39] Axie Infinity. Axie Infinity x Defi: play to earn is evolving!, May 2020. URL https://axieinfinity.medium.com/axie-infinity-x-defi-play-to-earn-is-evolving-23568aebc7ee. pages 16

[40] Axie Infinity. Breeding and $SLP, November 2021. URL https://whitepaper.axieinfinity.com/gameplay/breeding. pages 16

[41] Upland. Upland 101: how to play, December 2019. URL https://medium.com/upland/upland-101-how-to-play-b299a2bfd497. pages 17

[42] Upland. Uplandme, June 2022. URL https://play.upland.me/. pages 17

[43] Alien Worlds. Welcome to the Alien Worlds metaverse!, April 2021. URL https://alienworlds.medium.com/welcome-to-the-alien-worlds-metaverse-27040124d6a2. pages 18

[44] Alien Worlds. Happy first anniversary Alien Worlds!, January 2022. URL https://alienworlds.medium.com/happy-first-anniversary-alien-worlds-6086423a7bcf. pages 18

[45] Alien Worlds. Art history & lore, March 2021. URL https://alienworlds.io/art-history-lore/. pages 18

[46] Brian Beers. What regression measures, October 2021. URL https://www.investopedia.com/terms/r/regression.asp. pages 20

[47] Seldon. Decision trees in machine learning explained, November 2021. URL https://www.seldon.io/decision-trees-in-machine-learning. pages 20

[48] scikit-learn. 1. 10. Decision trees, 2022. URL https://scikit-learn/stable/modules/tree.html. pages 20

[49] IBM Cloud Education. What are neural networks?, August 2020. URL https://www.ibm.com/uk-en/cloud/learn/neural-networks. pages 20

[50] Mehreen Saeed. A gentle introduction to sigmoid function, August 2021. URL https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/. pages 21

[51] Danqing Liu. A practical guide to ReLU, November 2017. URL https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7. pages 21

[52] IBM Cloud Education. What are recurrent neural networks?, September 2020. URL https://www.ibm.com/cloud/learn/recurrent-neural-networks. pages 21

[53] IBM. Mean squared error, January 2022. URL https://www.ibm.com/docs/en/cloud-paks/cp-data/3.5.0?topic=overview-mean-squared-error. pages 22

[54] IBM. R2, April 2021. URL https://www.ibm.com/docs/en/cognos-analytics/11.1.0?topic=terms-r2. pages 22

[55] Aran Davies. What are the 5 best smart contract platforms for 2022?, 2022. URL https://www.devteam.space/blog/what-are-the-5-best-smart-contract-platforms-for-2022/. pages 23

[56] Kai Sedgwick. Report claims 34,000 Ethereum smart contracts are vulnerable to bugs, February 2018. URL https://news.bitcoin.com/report-claims-34000-ethereum-smart-contracts-vulnerable-bugs/. pages 23

[57] Martin Young. Reliably unreliable: Solana price dives after latest network outage, June 2022. URL https://cointelegraph.com/news/reliably-unreliable-solana-price-dives-after-latest-network-outage. pages 23

[58] Sebastian Sinclair. Cardano gains smart contract capability following 'Alonzo' hard fork, September 2021. URL https://www.coindesk.com/tech/2021/09/12/cardano-gains-smart-contract-capability-following-alonzo-hard-fork/. pages 23

[59] Stefan Stankovic. Cardano's smart contracts face major scalability issue, September 2021. URL https://cryptobriefing.com/cardanos-smart-contracts-face-major-scalability-issue/. pages 23

[60] Kamil Polak. Hack solidity: block timestamp manipulation, January 2022. URL https://dev.to/kamilpolak/hack-solidity-block-timestamp-manipulation-1mbg. pages 25

[61] ConsenSys Diligence. Ethereum smart contract best practices, April 2022. URL https://consensys.github.io/smart-contract-best-practices/. pages 25, 34, 35, 36

[62] Steve Marx. Stop using solidity's transfer() now, September 2019. URL https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/. pages 25

[63] Freepik. Freepik | graphic resources for everyone, June 2022. URL https://www.freepik.com. pages 27

[64] Buns. On the GovernorDAO treasury fund, December 2020. URL https://learn-solidity.com/on-the-governor-dao-treasury-fund-13d3525d5682. pages 31

[65] David Rodeck and Mark Hooson. Top NFT marketplaces of 2022, May 2022. URL https://www.forbes.com/uk/advisor/investing/best-nft-marketplaces/. pages 31

[66] OpenSea. ERC721 tutorial, April 2022. URL https://docs.opensea.io/docs/getting-started. pages 31

[67] Etherscan. Null Address, June 2022. URL https://etherscan.io/address/0x0000000000000000000000000000000000000000. pages 36

[68] Vitalik Buterin. How do I make my dapp ”serenity-proof?”, January 2016. URL https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof/200#200. pages 36

[69] Lucas Aschenbach. 8 ways of reducing the gas consumption of your smart contracts, February 2019. URL https://medium.com/coinmonks/8-ways-of-reducing-the-gas-consumption-of-your-smart-contracts-9a506b339c0a. pages 36, 52

[70] MongoDB. Why use MongoDB and when to use it?, 2022. URL https://www.mongodb.com/why-use-mongodb. pages 39

[71] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. pages 42

[72] François Chollet et al. Keras. https://keras.io, 2015. pages 42

[73] Prettier. Prettier, June 2022. URL https://github.com/prettier/prettier. pages 43

[74] Hideo Hattori. Autopep8: a tool that automatically formats python code to conform to the pep 8 style guide, October 2021. URL https://github.com/hhatto/autopep8. pages 43

[75] Nicholas C. Zakas and Ilya Volodin. ESLint, June 2022. URL https://github.com/eslint/eslint. pages 43

[76] Python Code Quality Authority. Pylint: python code static checker, June 2022. URL https://www.pylint.org/. pages 43

[77] Protofire. Solhint, May 2021. URL https://github.com/protofire/solhint. pages 43

[78] Ethereum. NatSpec Format — Solidity 0.8.15 documentation, February 2022. URL https://docs.soliditylang.org/en/latest/natspec-format.html. pages 43

[79] Google. Google Python style guide, June 2022. URL https://google.github.io/styleguide/pyguide.html. pages 43

[80] Scott Chacon and Ben Straub. *Pro Git*. Apress, second edition, April 2022. pages 43

[81] Hardhat. Hardhat | Ethereum development environment for professionals by Nomic Foundation, 2022. URL https://hardhat.org. pages 44

[82] Mocha. Mocha - the fun, simple, flexible JavaScript test framework, 2022. URL https://mochajs.org/. pages 44

[83] Jest. Jest delightful javascript testing, 2022. URL https://jestjs.io/. pages 44

[84] GitHub. GitHub actions, 2022. URL https://github.com/features/actions. pages 44

[85] Emilio López, Feist Josselin, and Gustavo Grieco. Echidna-action, January 2018. URL https://github.com/crytic/echidna-action. pages 44

[86] Feist Josselin and Emilio López. Slither-action, March 2022. URL https://github.com/crytic/slither-action. pages 44

[87] cgewecke and Alex Rea. Solidity-coverage, December 2019. URL https://www.npmjs.com/package/solidity-coverage. pages 47

[88] CoinMarketCap. Top play to earn tokens by market capitalization, June 2022. URL https://coinmarketcap.com/view/play-to-earn/. pages 47

[89] Charles Holtzkampf and Sentnl. Contract audit results. Audit, February 2021. URL https://github.com/Alien-Worlds/Security-Audits/blob/main/Smart%20contract%20Audit%20%20Results%20-%20Alienworlds-2.pdf. pages 48

[90] Quantstamp. Security assessment certificate Axie Infinity. Audit, October 2020. URL https://cdn.axieinfinity.com/landing-page/AXS_Audit_Report.pdf. pages 48

[91] Certik. ULAND - Certik security leaderboard, February 2022. URL https://www.certik.com/projects/uland. pages 48

[92] Certik. The Sandbox - Certik security leaderboard, January 2022. URL https://www.certik.com/projects/sandbox.game. pages 48

[93] cgewecke. Hardhat-gas-reporter, February 2022. URL https://github.com/cgewecke/hardhat-gas-reporter. pages 48

[94] Sky Mavis. The Ronin Block Explorer, June 2022. URL https://explorer.roninchain.com/. pages 48

[95] Binance Academy. What is a smart contract security audit?, March 2022. URL https://academy.binance.com/en/articles/what-is-a-smart-contract-security-audit. pages 53