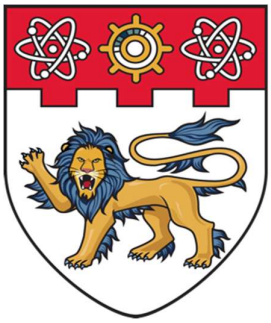

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Final Year Project

Quantitative Analysis of Smart Contracts (Fairness)

Final Report

Chan Shao Jing

U1921314E

CHAN0924@e.ntu.edu.sg

Supervisor: Ast/P Li Yi

Examiner: A/P Goh Wooi Boon

Abstract

Smart contracts are self-executing digital contracts that enables trust, transparency, and automation on various blockchains to facilitate transactions. However, ensuring fairness in smart contracts have always been a critical challenge *(Singh, 2019)* due to users' technical inability to analyse the contract code being executed. This project will illustrate and categorise existing malicious attacks on smart contracts into standards for evaluating fairness. These standards will then be used to analyse smart contract transactions in popular gaming decentralised applications. Our findings indicated that even smart contracts in the highest grossing applications are not perfect and contains flaws that could undermine their reliability.

Acknowledgments

I wish to express my gratitude to Ast/P Li Yi and Research Associate Liu Ye for their valuable contribution, guidance, and support throughout my final year project. Without their constant advice the past year, this project would have been impossible to complete. I am honoured to have the opportunity to work on this research project as my graduation thesis under the School of Computer Science and Engineering in Nanyang Technological University. Lastly, I am also thankful to A/P Goh Wooi Boon for taking the time to examine and grade this project.

Table of Contents

List of Figures

1 Introduction

1.1 Background

Blockchain technology has gained widespread attention and adoption since it was first introduced in 2008, with cryptocurrency being one of its most well-known use cases. According to the cryptocurrency tracking website CoinMarketCap, cryptocurrencies' total market capitalisation reached a total of $3 trillion USD with an average daily trading volume of $100 billion USD at its peak in 2022. Blockchain technology's prominence could be attributed to its appeal as a decentralised and secure alternative to traditional financial systems.

In 2015, the introduction of the 2nd most popular blockchain, Ethereum, allowed the development and implementation of smart contracts in decentralised applications (dApps). Smart contracts are self-executing contracts written in computer programmes that could execute automatically on the blockchain when certain conditions are met. The creation of smart contracts allowed users to transact on the blockchain without the need for intermediaries or trusted third parties to facilitate, verify or enforce the negotiation or performance of these contracts between parties.

As blockchain and smart contracts become widely popular, cybercrimes augment as well. With smart contracts being publicly available on blockchain, cyber criminals are constantly investigating them and on the hunt for money-making opportunities in the design of the system. They will often exploit users who do not have the technical knowledge to internalise the processes and source code being executed on the blockchain. There have also been increasingly publicised cases of cyber fugitives creating malicious smart contracts that aim to exploit and abuse participating parties in the smart contracts and cause damage to the trust and usage of the blockchain technology *(Sigalos, 2022).* Although it is on the cyber fugitives for exploiting these smart contracts' flaws, there should be equal responsibility on the developers to ensure their smart contracts are robust and impartial to participating parties.

1.2 Scope

As such, this project will be focusing on assessing the top grossing gaming decentralised applications, which often involves transactions of fungible and non-fungible tokens on the networks they are built upon respectively. The analysis emphasis will be on the possible vulnerabilities of the targeted dApps and ensuring these dApps are equitable to the parties involved in their smart contract transactions. Based on the analysis results, there will be a further discussion about fascinating and atypical points discovered in the quantitative analysis of these dApps' smart contracts.

1.3 Objectives

- To thoroughly understand blockchain technology and the reason behind why it is so popular and influential
- To shortlist analytical standards of fairness and popular gaming dApps available
- To conduct quantitative analysis on the dApps based on defined fairness standards
- To provide objective analysis on the findings and discuss results, while providing possible extensions and implementations of the project

1.4 Timeline

The following graph is a simple and generalised Gantt chart used to illustrate the project's timeline:

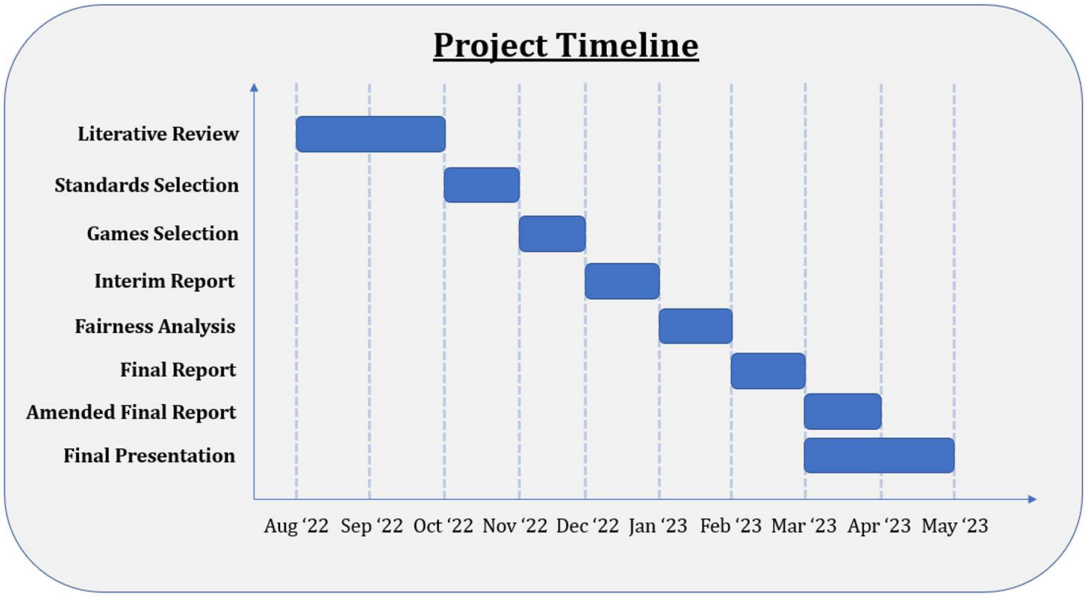

Figure 1: Project Timeline

1.5 Applications

The project will require the usage of the following external software applications to run and document work completed:

- Visual Studio Code
- Github
- Solidity
- Microsoft Word
- Teams

2 Literature Review

2.1 Blockchain

The idea of a blockchain concept was first introduced back in a 1982 dissertation by cryptographer David Chaum. In his paper, Chaum discussed how transactions could be secured by recording the transaction history in network nodes via cryptography *(Chaum, 1982).* He further released a follow up research paper in 1983 on the concept of blind signatures, a form of digital signatures that securely conceal the details of a message before it is signed by the receiver *(Chaum, 1983)*. This concept broke down the walls of how traditional transactions are executed and gave rise to the establishment of digital currency, which allows anonymous and secure e-transactions.

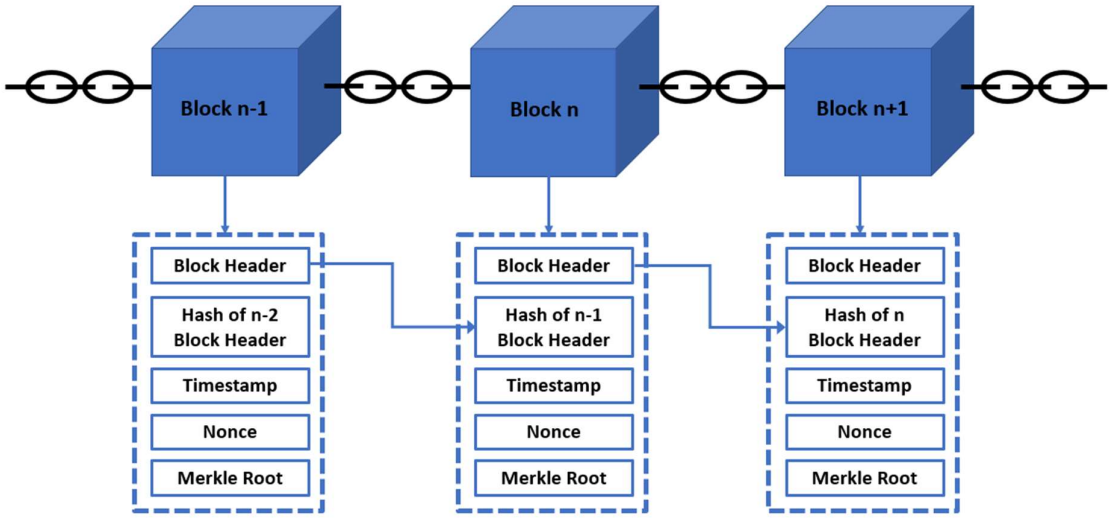

Figure 2: General Structure of a Blockchain

Blockchains utilise the Distributed Ledger Technology which allows concurrent access, validation and record updating across the networked database. The following is a list of information available in each block of the blockchain:

- **Block Header:** The block header is the unique identity of each block and is hashed periodically by miners to create proof of work for mining rewards. Proof of work is a decentralised consensus mechanism that is computed to verify cryptocurrency transactions and to add them to the blockchain. The block header will contain 3 sets of block metadata and multiple individual components.
- **Previous Block Address/Hash:** The hash contains the address of the previous block and is used to connect the previous block to the current block.
- **Timestamp:** Timestamp is a string of characters of the date and time that is assigned to digital documents when they are created. It can also be used to verify the data in the block.
- **Nonce:** Nonce is an artificially generated number that acts as a counter during mining process. Miners will constantly try to input nonce values until the hashing output falls below the predetermined threshold. When this happens, the block is considered mined, validated, and added to the blockchain. As such, each block's nonce is only used once.
- **Merkle Root:** The Merkle Root is the start of the Merkle Tree which contains all information about every single transaction that exists on the current block. Merkle Root is also the single digital fingerprint hash value that any transaction on the block can use to validate their data against.

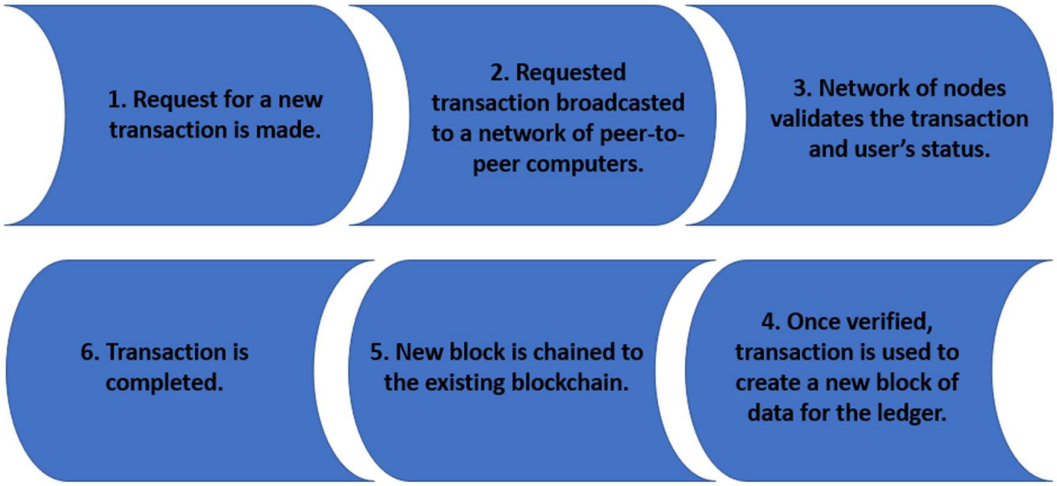

Figure 3: Workflow for a New Transaction

For a new transaction to be added to any blockchain, it must undergo a workflow illustrated above. The request of a new transaction will be transmitted to seek validation from a network of peer-to-peer computers scattered across the world. These computers will verify both the transaction and the user's status using known algorithms for computation. Approval replies are sent to the blockchain to create a new block using data of the new transaction. The new block created is then chained to the existing blockchain, remaining there forever and totally immutable.

The above workflow will allow blockchains to function securely without the need of an intermediary for ensuring the validity and integrity of transactions and data stored on the blockchain. Without a central authority, the risk of a single point of failure or hacking is greatly reduced, and the cohesion of the network is maintained through the collective efforts of its participants. Ultimately, the decentralisation of blockchain provides benefits such as increased security, transparency, and accountability.

2.2 Smart Contracts

Thereafter from 1994 to 1997, Nick Szabo, a computer scientist, published research papers which built on Chaum's concept and defined smart contracts as "contracts that facilitate, verify, and enforce negotiation or performance of a contract via digital means" *(Szabo, 1994).* Smart contracts are implemented using Chaum's digital signatures technology and introduced automation and enforceability to Chaum's blockchain concept.

It was not until 2008 where Satoshi Nakamoto (a pseudonym) published a revolutionary Bitcoin white paper that built on the past decades of research in blockchain and transformed blockchain and smart contracts from being just a discussed concept into reality. Satoshi outlined the concept of a peer-to-peer electronic cash system that would allow for the transfer of funds without the need for a central authority or intermediary, primarily through smart contracts *(Nakamoto, 2008).* Satoshi also created and published the first version of Bitcoin software and set up a Bitcoin community that continued developing Bitcoin in the years to come.

Ultimately, blockchain and cryptocurrency only became known worldwide when they experienced a spurt of rising popularity during 2013 to 2017, mainly when the price of Bitcoin and other cryptocurrencies began to rise. The blockchain technology received mainstream attention as investors and companies started noticing the possible usage of digital currencies and centred their businesses' operations around it.

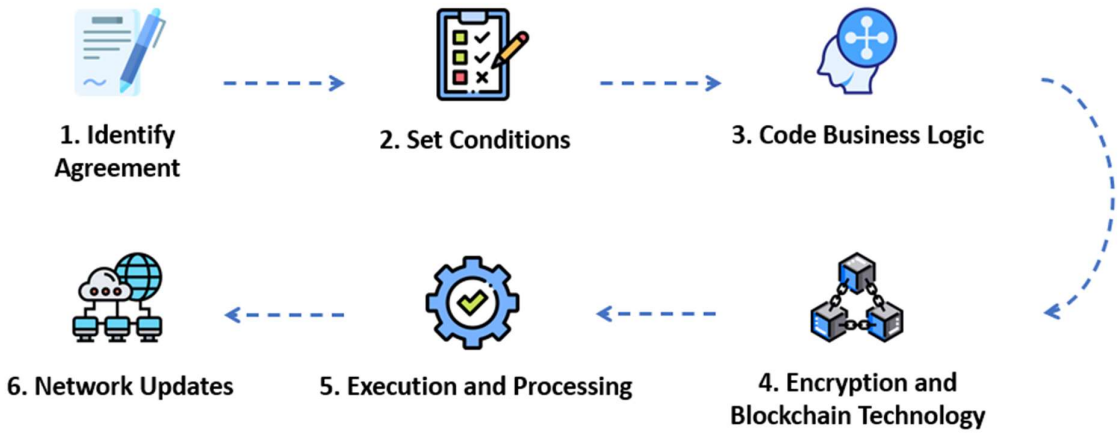

Figure 4: How Smart Contract Works

1. **Identify Agreement:** Several parties come together to recognize a collaborative opportunity and define their desired objectives. The agreements reached between them may involve various activities, such as the exchange of assets, execution of business processes, and so on.
2. **Set Conditions:** Smart contracts can be triggered either by the involved parties themselves or automatically when certain predefined conditions are fulfilled. These conditions could be based on a variety of factors such as financial market indicators, specific events like GPS locations, and so on.
3. **Code Business Logic:** A program is created to automatically execute when certain conditions are fulfilled.
4. **Encryption and Blockchain Technology:** Encryption enables the secure verification and transmission of messages between the parties involved in smart contracts.

5. **Execution and Processing:** In the blockchain process, once there is agreement between the parties on authentication and verification, the code is executed, and the results are recorded for verification and compliance purposes.
6. **Network Updates:** Once smart contracts are executed, all the nodes on the network update their ledger to reflect the latest state. When the record is added and validated on the blockchain network, the record will be unable to receive any alteration as it is in append-only mode.

2.3 Decentralised Applications

Ethereum is the 2nd most popular derivative of the blockchain technology (behind Bitcoin) and was first produced in 2013 by Vitalik Buterin, a Canadian programmer. Unlike Bitcoin, which primarily serves as a digital currency, Ethereum's focus is on providing a platform for dApps that use smart contracts. Ethereum has further advanced smart contracts functionalities by providing a more flexible and versatile platform for creating and executing smart contracts such as allowing smart contracts to be integrated into a frontend user interface. This allows developers to create more complex and sophisticated smart contracts that can execute a wide range of tasks, beyond just transferring digital currencies.

Ethereum's platform also introduced the use of Solidity, a programming language specifically designed for writing smart contracts, making it easier for developers to create and test their smart contracts. Ethereum has revolutionised smart contracts, allowing dApps to be created for a wide spectrum of purposes: decentralised learning platforms, social media platforms, marketplaces, prediction markets and even games. Universally, Ethereum's platform and tools have helped to expand the possibilities of what can be achieved with smart contracts, leading to increased interest and adoption of this technology.

2.3.1 Targeted Blockchains

After the Ethereum blockchain, countless blockchain descendants that incorporate dApps were introduced. The newer blockchains' smart contract code are being written in mainly 2 different formats: JSON format or in Solidity. Both formats have their advantages and disadvantages: Solidity is specifically designed for Ethereum blockchain and has a wide range of features and functionalities. While JSON is a widely used data format in web development and API integrations, it is not a programming language and not specifically designed for smart contracts. However, some blockchain platforms do use JSON for their smart contracts as JSON is a lightweight data format that is easily readable and writable by both humans and machines. Their developers will be able easily write, read, and parse the code, making it more accessible and interoperable across different systems and platforms. Arguably, Solidity is still more popular in smart contracts development as it has an extensive online community of contributors.

Nonetheless, as the aim of this project was to analyse source code to expose vulnerabilities, investigating JSON format smart contract codes will not amount to a fruitful analysis. Thus, this project had shortlisted the popular blockchains that are still writing smart contracts code in Solidity format (right column in table below).

JSON	**Solidity**
• WAX • Hive • EOS • Ronin • Flow • Klaytn	• Ethereum (ETH) • Polygon • BNB Chain

Figure 5: Divide of Blockchain Formatting

2.3.2 Targeted Decentralised Applications

With the focus on gaming dApps, this project had utilised DappRadar to filter the top grossing gaming dApps in December 2022. The ranking of dApps was based on the number of unique active wallets interacting with the dApp's smart contracts to ensure the produced analysis results would be beneficial to the masses. From the top 15 ranking list, a further manual filter was applied based on the dApps' blockchain networks, choosing only those that are built on blockchains with Solidity format support and available online (highlighted in the table below).

Rank (by UAW)	Title	Networks	Balance ($)	UAW	Volume
1	Alien Worlds	• WAX • BNB Chain	2.89M	590.37k	6.37M
2	Splinterlands	• Hive • WAX	47.96K	326.96k	32.58k
3	Planet IX	• Polygon	69.53M	157.42k	3.54M
4	Benji Bananas	• ETH • Polygon	28.48M	125.02k	888.94
5	Upland	• EOS	42.17K	119.62k	0
6	Farmers World	• WAX	0	116.17k	14.73k
7	Axie Infinity	• Ronin • ETH	773.72M	108.47k	73.65M
8	Oath of Peak	• Polygon	9.78	86.04k	34.29k
9	Trickshot Blitz	• Flow	0	84.18k	28.17k
10	Iskra	• Klaytn • ETH	52k	53.5k	101.65k
11	Arc8 by GAMEE	• Polygon	43.12M	51.33k	603.83
12	Era7: Game of Truth	• BNB Chain	14.66M	42.69k	160.31k
13	Playzap Games	• BNB Chain	546.9	42.19k	243.08
14	Meta Apes	• BNB Chain	53.89k	39.33k	1.67M
15	Sunflower Land	• Polygon	2.91k	36.46k	70.53k

Balance: Total value of assets in dApp's smart contracts

UAW: Number of unique active wallets interacting with dApp's smart contracts

Volume: Total amount of incoming value to dApp's smart contracts

Note: Although 8. Oath of Peak is built on Polygon blockchain, it was omitted as it is a special multiplayer card game that has its source code securely unavailable.

Figure 6: Targeted Decentralised Applications

2.4 Fairness Standards

Fairness standards are a set of criteria used to evaluate the fairness of smart contracts in dApps. The aim was to ensure that smart contracts were not biased against any particular participating party and that all parties involved in any dApp smart contract were treated fairly. To assess the fairness of a dApp, every of the dApp's smart contracts were evaluated against these criteria and the results were used to identify any potential issues and to improve the design and implementation of the contract.

2.4.1 Execution Standard

The execution standard of a smart contract refers to the rules and procedures that dictate how a smart contract is executed and enforced on a blockchain. This includes factors such as the programming language used to write the contract, the cryptographic algorithms used to secure it, and the consensus mechanism used to validate and execute the contract on the network. The execution standard ensures that the smart contract operates as intended, with clear and transparent rules and procedures for all parties involved. It is a critical aspect of the reliability and trustworthiness of dApps. Listed below are some of the common malicious attacks found that will violate the execution standard:

- Logic bugs: Smart contracts can contain bugs in their programming logic that allow attackers to exploit them. These types of bugs can be difficult to detect and can result in unexpected behaviour.

- Re-entrancy attacks: This occurs when a smart contract repeatedly calls back to a vulnerable contract, either causing it to run out of gas and potentially allowing the attacker to take control of the contract or to continuously steal tokens from the contract.
- Denial-of-Service (DoS) attacks: In this type of attack, an attacker floods the contract with excessive requests or transactions, overwhelming the system and causing it to slow down or fail completely.

2.4.2 Input Standard

The input standard of a smart contract refers to the format and type of data that is accepted as input by the contract. This includes parameters, arguments, and any other data that is required for the contract to execute properly. Input standards may vary depending on the programming language used to write the contract and the specific requirements of the contract. For Solidity smart contracts, input parameters are typically defined using data types such as uint (unsigned integer), string, or bool (boolean). The input standard is important to ensure that the contract can properly process and validate the data it receives, and to prevent errors or security vulnerabilities. Listed below are some of the common malicious attacks found that will violate the input standard:

- Integer overflow/underflow attacks: This happens when a smart contract fails to properly handle large or small numbers, allowing the numbers to exceed the maximum or minimum representable value of the data type used to store it. Attackers can exploit this by creating situations where unintended additional funds are being transferred, leading to security vulnerabilities and financial losses.
- Malicious inputs: An attacker can provide malicious inputs to a smart contract, causing it to behave unexpectedly or even crash.
- False data injections: An attacker can manipulate the data inputs of a smart contract to cause it to produce false results, potentially leading to attackers being able to manipulate the contract's variables and potentially steal funds.

An important note for integer overflow/underflow: from Solidity version 0.8.0 onwards, the compiler has default checks for all arithmetic calculations. The compiler will revert automatically in cases of integer overflow and underflow. Thus, there is no longer a need to check for such edge cases. However, for Solidity version before <0.8.0, most dApps have implemented checking using a SafeMath library written by the community, but the versions written vary depending on the dApp. Otherwise, the dApps will use the *require* keyword in Solidity to enforce values after arithmetic calculations, though the fairness analysis discussed later will show not all enforcements had been comprehensive enough.

2.4.3 Timing Standard

The timing standard of a smart contract refers to the rules and requirements for the timing of certain actions or events within the contract. This can include the timing of when the contract becomes active, when specific actions must be taken, and when certain conditions must be met for the contract to execute. Timing standards can help ensure that smart contracts operate effectively and efficiently, and that all parties involved have a clear understanding of when actions and events will occur. Listed below are some of the common malicious attacks found that will violate the timing standard:

- Time manipulation attacks: An attacker can manipulate the timestamps or block times in a smart contract, allowing them to manipulate the outcome of time-dependent actions, such as auctions or lotteries.
- Front-running attacks: An attacker can manipulate the order in which transactions are processed, allowing them to take advantage of price differences or steal funds from other users.
- Side-channel attacks: These types of attacks involve monitoring the timing of messages sent between nodes on the network to infer and exploit information unintentionally leaked during the execution of the protocol or contract, such as timing information or power consumption.

2.4.4 Integrity Standard

The integrity standard of a smart contract refers to the requirement that the contract's code and data remain unchanged and uncorrupted throughout its execution. This means that the smart contract should not be susceptible to tampering, hacking, or other malicious activities that could compromise its integrity. During development of smart contract code, smart contract developers may use various techniques such as code reviews, testing, and auditing to identify and address potential vulnerabilities and ensure that the contract's integrity is maintained. Listed below are some of the common malicious attacks found that will violate the integrity standard:

- Phishing attacks: Attackers can use social engineering techniques to manipulate users into making decisions that benefit the attacker or create fake smart contracts that look like legitimate ones, convincing users to approve a malicious transaction or disclose sensitive information.
- Insider attacks: These occur when an individual with privileged access or knowledge of a smart contract exploits that access to manipulate the contract for personal gain.
- Authentication attacks: In this type of attack, an attacker gains access to a smart contract's private key, allowing them to manipulate the contract's code or steal funds.

3 Fairness Analysis

In each subsection, the report will give a summary of the dApp game, detailing the objectives and tokens involved in each game. Thereafter, there will be an overview map of the smart contracts involved in each game modelled based on their source code and a discussion of whether the game satisfies each fairness standard. If a game violates a certain fairness standard, the report will elaborate and justify with examples.

3.1 Alien Worlds

Alien Worlds is a dApp game built on the WAX and BNB blockchains. In the game, players can buy, sell, and trade virtual land on different planets, and use their land to mine digital assets and earn cryptocurrency. The game also features a governance system, where players can vote on proposals and decisions related to the game's development and management. Alien Worlds uses non-fungible tokens as game assets, which are stored on the blockchains and can be traded on various marketplaces.

Smart Contracts Overview

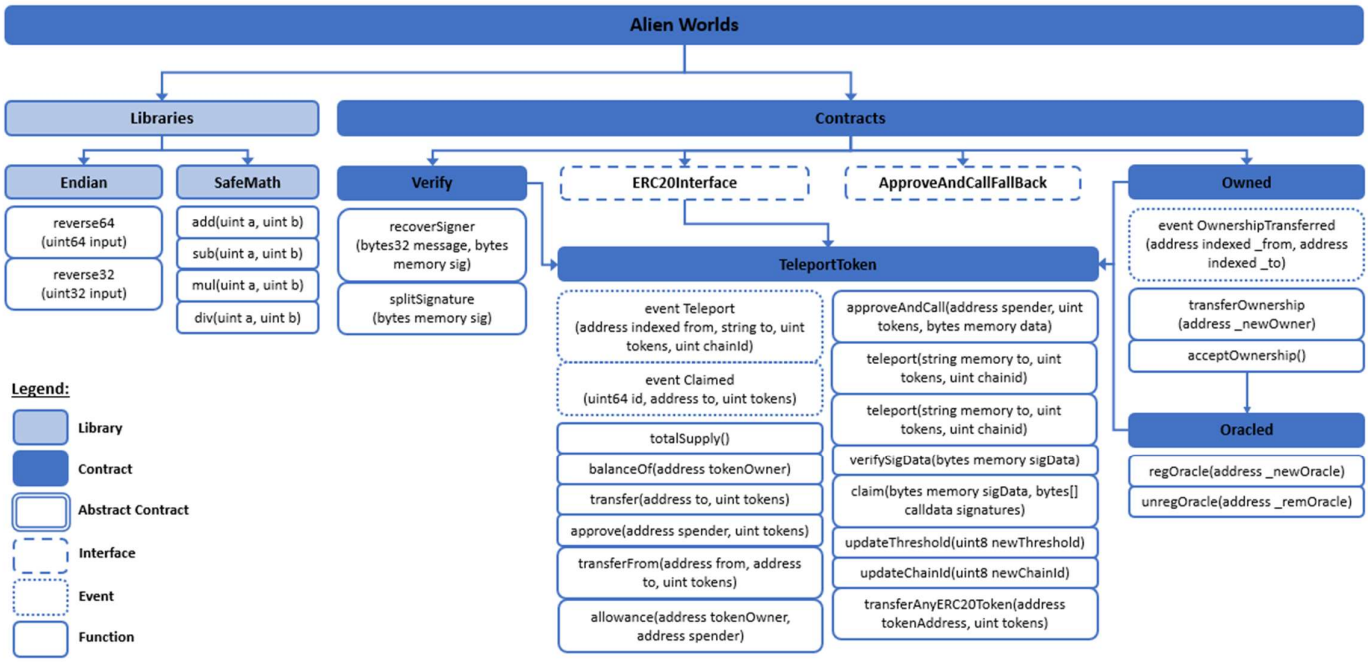

Figure 7: Overview of Alien Worlds' Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Alien Worlds	Violates	Satisfies	Violates	Satisfies

Figure 8: Summary of Alien Worlds' Fairness Analysis

Execution Standard Violation:

2 of Alien Worlds' functions, transfer() and transferFrom(), to transfer tokens in the TeleportToken contract were found to be missing sender checking. An adversary will be able to exploit this flaw by launching the Denial-of-Service (DoS) attack.

```
function transfer(address to, uint tokens) override public returns (bool success) {
    balances[msg.sender] = balances[msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(msg.sender, to, tokens);
    return true;
}
```

```
function transferFrom(address from, address to, uint tokens) override public returns (bool success) {
    balances[from] = balances[from].sub(tokens);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(from, to, tokens);
    return true;
}
```

Figure 9: transfer() and transferFrom() Source Code

As example, an adversary Annie can trigger the transfer function through any in-game contract that involves them, sending 1000 tokens from her address back to her own address. Since there is no net change in tokens of her address, Annie can ramp up and repeat this process 1000 times per second through an automated script, causing the system traffic to be heavily congested or even fail completely. Annie can also easily get hold of information such as timing of in-game events, timing her attacks along with the

schedule and causing massive mayhem in the reliability and trust of the game's transactions.

```
function transfer(address to, uint tokens) override public returns (bool success) {
    require(msg.sender != to, "Sender and receipient should not be the same address!");
    balances[msg.sender] = balances[msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(msg.sender, to, tokens);
    return true;
}
```

```
function transferFrom(address from, address to, uint tokens) override public returns (bool success) {
    require(from != to, "Sender and receipient should not be the same address!");
    balances[from] = balances[from].sub(tokens);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(from, to, tokens);
    return true;
}
```

Figure 10: transfer() and transferFrom() Corrected Code

Alien Worlds' developers should include a line of code to check and revert the contract if the sender and receiver is from the same address. This would eliminate the chances of Annie successfully conducting a DoS attack.

Timing Standard Violation:

The ownership acceptance function, acceptOwnership(), in the Owned contract has the event OwnershipTransferred emitted before the actual ownership is transferred. An adversary will be able to make use of the time difference between the successful event log and actual ownership transfer to launch a front-running attack on this contract.

```
function acceptOwnership() public {
    require(msg.sender == newOwner);
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
    newOwner = address(0);
}
```

Figure 11: acceptOwnership() Source Code

For example, the adversary Annie owns an Alien Worlds planet. The player Peter has a transaction lined up with a pre-set condition such that when the event log shows that another player Paul acquired the ownership of a planet, Peter will transfer 1000 tokens to the planet's address in exchange for building at the planet. Annie attained this insider knowledge and engaged Paul, ultimately selling the planet to Paul. Annie also previously set the planet's tokens withdrawal to be automatic. Peter's transaction of 1000 tokens will go through at the emitted OwnershipTransferred event log, transferring 1000 tokens to the planet. Since Annie is still the planet's owner, she obtains and withdraws the 1000 tokens. The next line of code in the contract is executed and Paul becomes the new planet owner but will not receive any token for Peter's building.

```
function acceptOwnership() public {
    require(msg.sender == newOwner);
    originalOwner = owner;
    owner = newOwner;
    emit OwnershipTransferred(originalOwner, newOwner);
    newOwner = address(0);
}
```

Figure 12: acceptOwnership() Corrected Code

Alien Worlds' developers should swap the 2 lines of code, allowing the transfer of ownership to go through before emitting the OwnershipTransferred event. This change will remove any ambiguity in transferring ownerships of Alien Worlds' tokens and allow subsequent transactions that depend on these ownership transfers to be executed accurately.

3.2 Arc8 by GAMEE

Arc8 is a dApp game built on the Polygon network. The game application itself contains several mini arcade games such as Hoop Shot, Pirate Solitaire, ATARI Asteroids etc. Most of the games are played competitively in 1v1 format against other players to win rewards in the form of cryptocurrency. Arc8 exclusively uses the GMEE token as the access and governance token to power Arc8's gaming platform.

Smart Contracts Overview

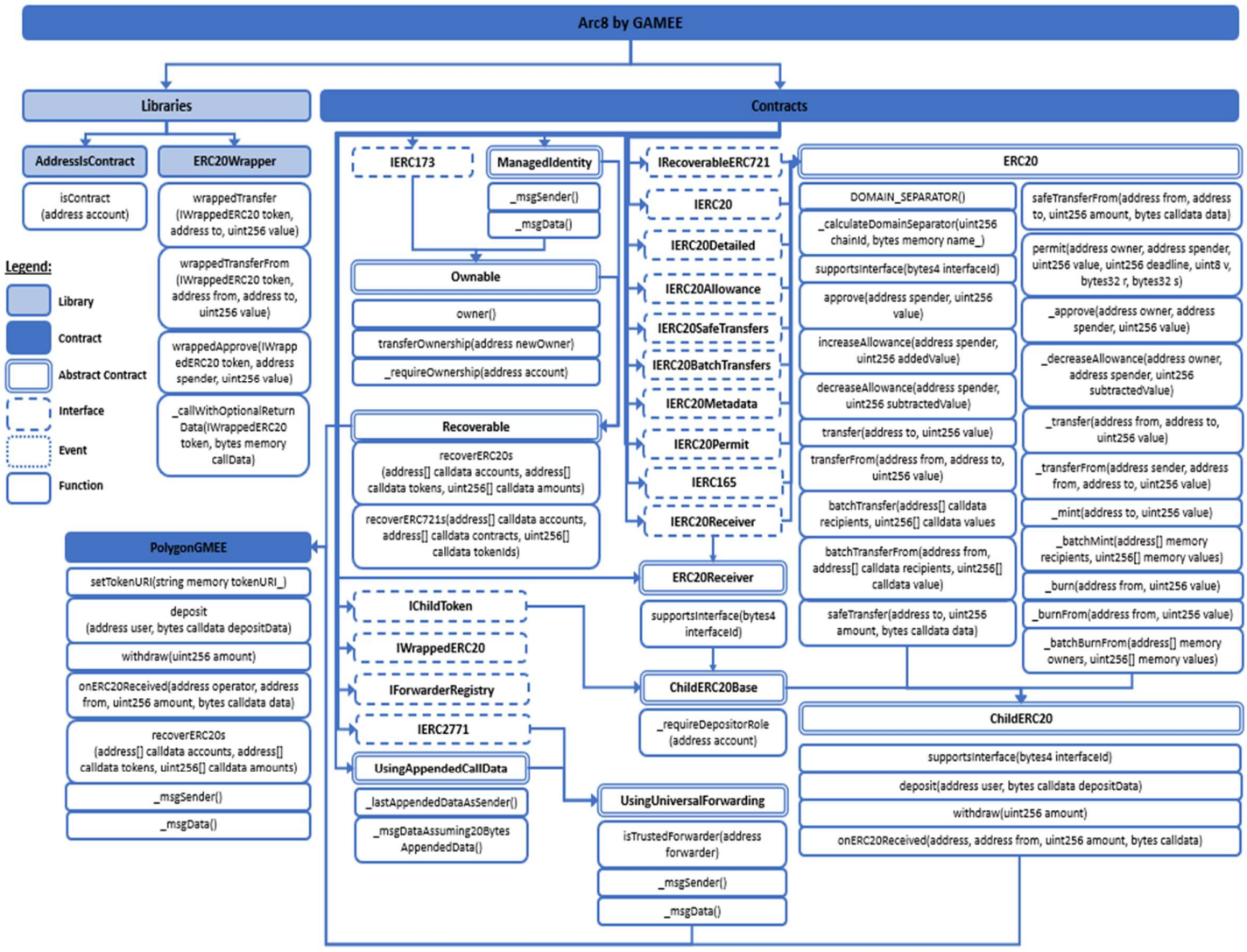

Figure 13: Overview of Arc8's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Arc8 by GAMEE	Satisfies	Violates	Violates	Satisfies

Figure 14: Summary of Arc8's Fairness Analysis

Input Standard Violation:

The deposit() and withdraw() functions in Arc8's PolygonGMEE contract do not account for possible integer overflow or underflow. Since Arc8's contract specifies the use of Solidity versions of >=0.7.6 <0.8.0 to compile, the automatic in-built integer overflow or underflow checks by Solidity are not in place. Arc8 also do not have the usual SafeMath library functions to conduct appropriate checks. Thus, an adversary can launch an integer overflow or underflow attack on this contract.

```
pragma solidity >=0.7.6 <0.8.0;
contract PolygonGMEE is Recoverable, UsingUniversalForwarding, ChildERC20 {
    using ERC20Wrapper for IWrappedERC20;

    uint256 public escrowed;
```

```
function deposit(address user, bytes calldata depositData) public virtual override {
    escrowed -= abi.decode(depositData, (uint256));
    super.deposit(user, depositData);
}

function withdraw(uint256 amount) public virtual override {
    escrowed += amount;
    super.withdraw(amount);
}
```

Figure 15: deposit() and withdraw() Source Code

For example, when the 'escrowed' variable is initialised as an unsigned integer data type of 256 bits and a current value of 0, the adversary Annie can pass a deposit input of 1 to cause an integer underflow. When this happens, the escrowed value will decrement from

the max value of (2^{256} – 1) and cause unexpected results in future transactions. On the other hand, when the 'escrowed' variable has a current value of (2^{256} – 1), which is the max storage size of uint256, Annie can also pass a withdraw input value of 1 to cause an integer overflow. When this happens, the escrowed value will increment from the minimum value of 0 and cause incorrect results in future transactions.

```
function deposit(address user, bytes calldata depositData) public virtual override {
    require(abi.decode(depositData, (uint256)) <= escrowed, "Provided a deposit value that will cause integer underflow!");
    escrowed -= abi.decode(depositData, (uint256));
    super.deposit(user, depositData);
}

function withdraw(uint256 amount) public virtual override {
    require(amount + escrowed >= escrowed, "Provided a withdraw value that will cause integer overflow!");
    escrowed += amount;
    super.withdraw(amount);
}
```

Figure 16: deposit() and withdraw() Corrected Code

Arc8's developers should include these 2 lines of code highlighted, that will ensure the escrowed value do not underflow below 0 in the deposit() function and do not overflow above the max storage size of uint256 in the withdraw() function. When the 'require' check fails, the transaction will revert and send out the error message attached in the error log on the blockchain.

Timing Standard Violation:

When the approve() and transferFrom() functions in the ERC20 contract are used concurrently for transactions, there is a loophole where this combination will create a race condition situation, resulting in the adversary being able to transfer more tokens than the owner wanted to transfer. An adversary will be able to launch a front-running attack to steal funds from another user.

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

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

    _allowances[user][spender] = value;
    emit Approval(user, spender, value);
}
```

Figure 17: approve() Source Code

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

Figure 18: transferFrom() Source Code

For example, the player Paul allows the adversary Annie to transfer 10 tokens using the approve() function on a contract, passing in the arguments of Annie's address and value of 10. For some reason, Paul decided to change the allowed tokens from 10 to 20 and thus, calls the approve() function again with the new arguments of Annie's address and value of 20. Annie happened to notice Paul's second transaction before it was mined on the blockchain and decided to launch an attack by using the transferFrom() function to transfer Paul's tokens to her own wallet address, prioritizing this transaction by paying hefty priority fee. If Annie was able to execute her transaction successfully before Paul's second transaction, then Annie will have obtained Paul's initial 10 tokens and the new updated ability to transfer 20 of Paul's tokens after Paul's second transaction is finally executed. Annie can now quickly transfer the remaining 20 of Paul's tokens to her own wallet address, stealing a total of 30 tokens from Paul when she was supposed to be allowed to transfer 20 tokens only.

The main root cause of this race condition issue is due to the approve() method overwriting the current user's allowance without checking if the spender already used the allowance or not. The only exception being the specific scenario where the token owner is not an account, but a smart contract which can perform several operations atomically.

One simple workaround would be for users to always change transfer allowance to 0 first, making sure the change transaction is executed successfully before changing the transfer allowance to the value the user wanted. (e.g., 10 to 0 then 0 to 20) However, it will prove difficult to inform every user to conduct such checking for every transaction and could be challenging to analyse changes in internal transactions via the standard Web3 API.

```
function approve(address spender, uint256 value, uint256 currentValue) public returns (bool) {
    bool success = _approve(msg.sender, spender, value, currentValue);
    return success;
}
```

```
function _approve(address user, address spender, uint256 value, uint256 currentValue) internal returns (bool) {
    require(user != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");

    if (currentValue == _allowances[user][spender]) {
        _allowances[user][spender] = value;
        emit Approval(user, spender, value);
        return true;
    } else {
        return false;
    }
}
```

Figure 19: approve() Corrected Code

The developer's workaround will be to change the approve() and underlying computational _approve() functions. For approve(), we will pass in an additional argument currentValue from the user which is the current allowance given to this contract. The function will also be receiving a Boolean success result from the _approve() function call, returning the success result to upstream function call. For the _approve() function, we will also be receiving an additional argument of currentValue with a Boolean return value defined. There will be also an if-else loop structure, overwriting the current allowance only if the allowance is still the same as what was originally allocated,

currentValue. In this case, a Boolean true will be returned on successful allowance overwriting and a Boolean false otherwise.

Using the same Paul and Annie example, when Paul changes his token transfer allowance from 10 to 20, the overwriting would fail if Annie had transferred out the initial 10 tokens. A Boolean false will be returned and Paul would know that Annie had already transferred some if not all the tokens out to her own wallet address.

3.3 Axie Infinity

Axie Infinity is a dApp game built on the Ronin and Ethereum blockchains that allows players to collect, breed, and battle fantasy creatures called Axies. The game uses a cryptocurrency called "AXS" and "SLP" to incentivize gameplay and reward players for their participation. Players can earn rewards by winning battles and completing quests, which they can then use to purchase more Axies or trade on cryptocurrency exchanges.

Smart Contracts Overview

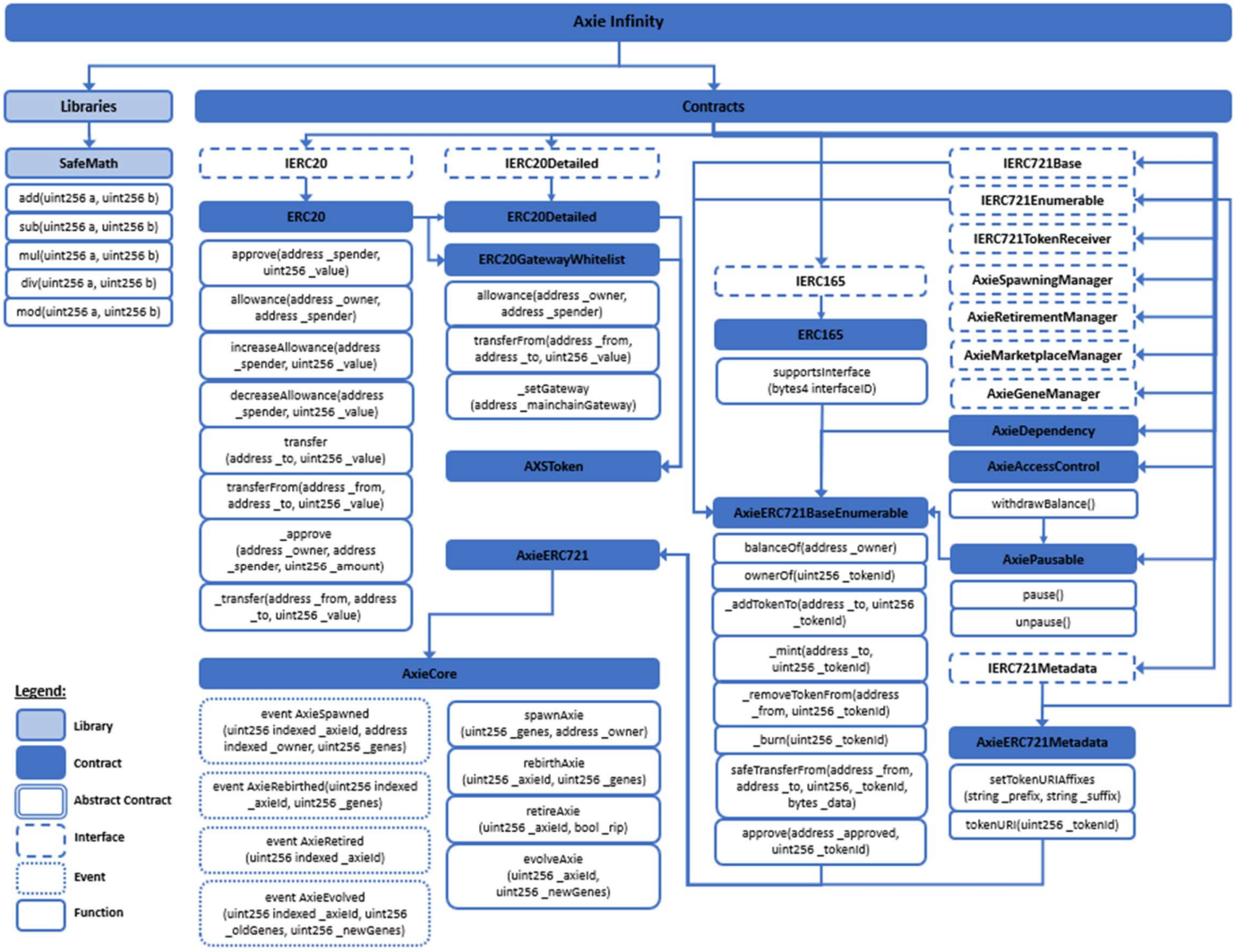

Figure 20: Overview of Axie Infinity's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Axie Infinity	Satisfies	Satisfies	Violates	Satisfies

Figure 21: Summary of Axie Infinity Fairness Analysis

Timing Standard Violation:

Axie Infinity also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.4 Benji Bananas

Benji Bananas is a free-to-play mobile game built on the Ethereum and Polygon blockchains. It is an adventure game which players must make Benji the monkey and other in-game characters leap through the jungle on vines to collect bananas for upgrades, specials, and power-ups. A Benji Bananas Membership Pass NFT purchase is required for players to unlock special PRIMATE tokens, which can then be exchanged for ApeCoin and other coins in the developer company Animoca Brands' ecosystem.

Smart Contracts Overview

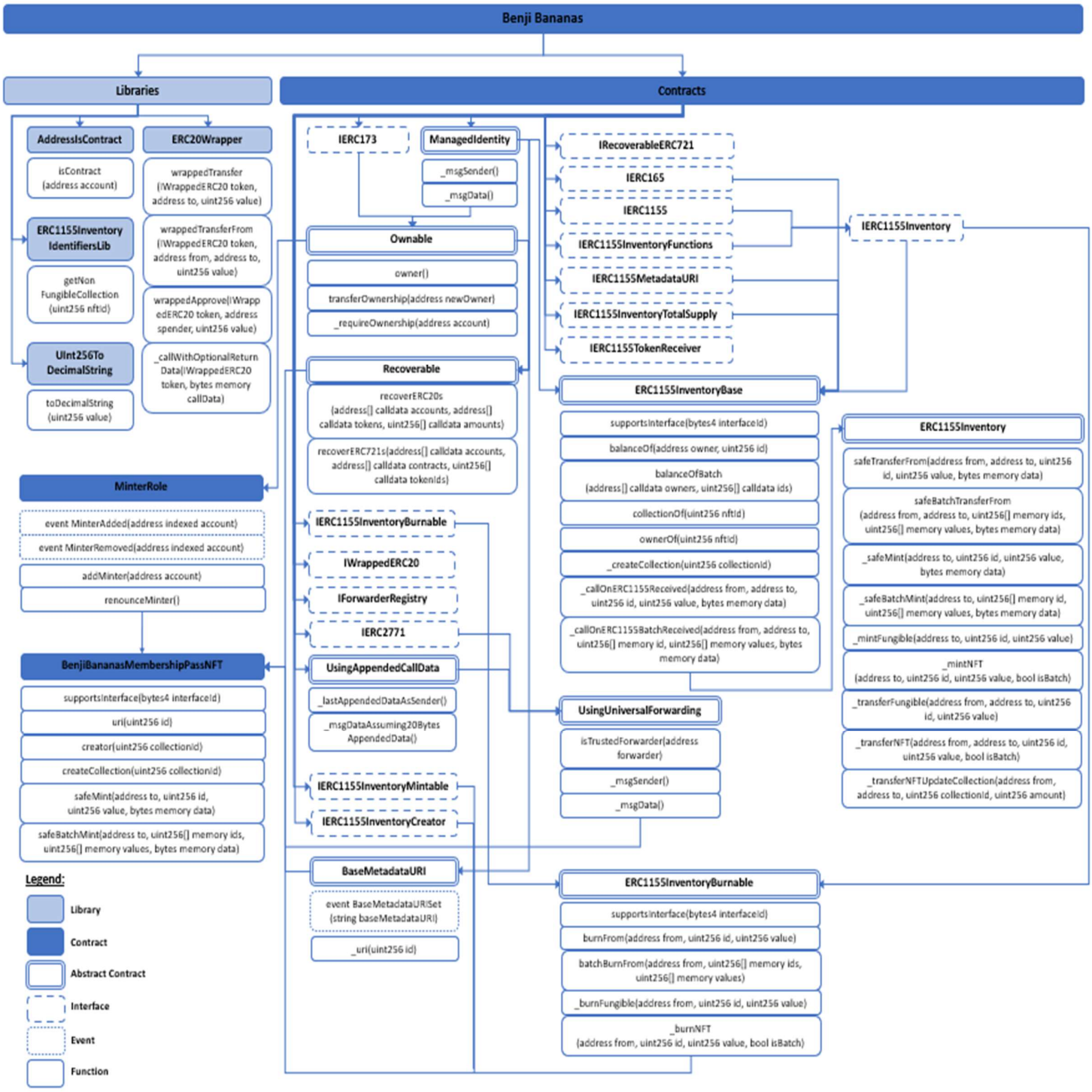

Figure 22: Overview of Benji Bananas' Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Benji Bananas	Violates	Satisfies	Satisfies	Satisfies

Figure 23: Summary of Benji Bananas Fairness Analysis

Execution Standard Violation:

Benji Bananas contains 3 functions with similar pitfall of using input arguments as the looping variable to change contract state variables. Since changing contract state variables consumes gas, the execution of these contracts could cause gas issues such as hitting the transaction gas limit or hitting the block's gas limit. They are namely batchBurnFrom() in the ERC1155InventoryBurnable contract, _safeBatchMint() and _safeBatchTransferFrom in the ERC1155Inventory contract. The analysis will use the batchBurnFrom() function to illustrate the logic bug.

```
function batchBurnFrom(
    address from,
    uint256[] memory ids,
    uint256[] memory values
) public virtual override {
    uint256 length = ids.length;
    require(length == values.length, "Inventory: inconsistent arrays");

    address sender = _msgSender();
    require(_isOperatable(from, sender), "Inventory: non-approved sender");

    uint256 nfCollectionId;
    uint256 nfCollectionCount;
    for (uint256 i; i != length; ++i) {
        uint256 id = ids[i];
        uint256 value = values[i];
        if (id.isFungibleToken()) {
            _burnFungible(from, id, value);
        } else if (id.isNonFungibleToken()) {
            _burnNFT(from, id, value, true);
            uint256 nextCollectionId = id.getNonFungibleCollection();
            if (nfCollectionId == 0) {
                nfCollectionId = nextCollectionId;
                nfCollectionCount = 1;
            } else {
                if (nextCollectionId != nfCollectionId) {
                    _balances[nfCollectionId][from] -= nfCollectionCount;
                    _supplies[nfCollectionId] -= nfCollectionCount;
                    nfCollectionId = nextCollectionId;
                    nfCollectionCount = 1;
                } else {
                    ++nfCollectionCount;
                }
```

Figure 24: batchBurnFrom() Source Code

```
function _burnFungible(
    address from,
    uint256 id,
    uint256 value
) internal {
    require(value != 0, "Inventory: zero value");
    uint256 balance = _balances[id][from];
    require(balance >= value, "Inventory: not enough balance");
    _balances[id][from] = balance - value;
    // Cannot underflow
    _supplies[id] -= value;
}
```

```
function _burnNFT(
    address from,
    uint256 id,
    uint256 value,
    bool isBatch
) internal {
    require(value == 1, "Inventory: wrong NFT value");
    require(from == address(uint160(_owners[id])), "Inventory: non-owned NFT");
    _owners[id] = _BURNT_NFT_OWNER;

    if (!isBatch) {
        uint256 collectionId = id.getNonFungibleCollection();
        // cannot underflow as balance is confirmed through ownership
        --_balances[collectionId][from];
        // Cannot underflow
        --_supplies[collectionId];
    }
}
```

Figure 25: _burnFungible() and _burnNFT Source Code

For example, assume that the player Paul wants to burn 601 tokens and the cost of burning 1 fungible or non-fungible token costs 50,000 units of gas at the base fee of 10 gwei (10^{-9} ETH) for the current empty block with capacity of 30,000,000 units of gas. In total, Paul's transaction should cost him 30,500,000 units of gas (601 tokens * 50,000 units of gas/token) or 300,500,000 gwei (601 tokens * 50,000 units of gas/token * 10 gwei/unit of gas) which will be translated to 0.3005 ETH (~$600 SGD).

For the first scenario, suppose Paul predicted this transaction to be costly but did not have the exact numbers for the usage. Paul made an estimation and assigned the gas limit

for this transaction to be 20,000,000 units of gas. Upon execution of the 400th token, the transaction gas limit is hit. According to the logic of the contract, the transaction runs out of gas and reverts the transaction. However, since the gas was already consumed by the miner for the computation of the contract, the gas is not returned to Paul. In addition to the failed transaction, Paul receives a loss of 200,000,000 gwei which translates to 0.2 ETH (~$400 SGD).

For the second scenario, assume that Paul learnt his lesson and assigned the transaction with the current maximum gas capacity of the Ethereum blockchain, 30,000,000 units of gas. Upon execution of the 600th token, the block's gas limit is hit. Like the first scenario, Paul's transaction reverts with 0 units of gas left, leaving Paul with another failed transaction and a loss of 300,000,000 gwei which translates to 0.3 ETH (~$600 SGD).

There are several possible methods that could increase Paul's probability of his transaction succeeding. Firstly, Paul could include a priority fee for his transaction, giving miners a higher incentive to mine his transaction before other transactions in the same block, depending on the amount of priority fee given. This method allows Paul's transaction to experience a larger proportion of the leftover block gas limit as his transaction will be executed first. Though, this means that Paul will be paying a higher cost to execute this transaction. Secondly, Paul can also benchmark his estimation of the gas limit for his transaction based on prior transactions in the block. Lastly, Paul can also split up his transaction into several smaller transactions, though this means he will now have to estimate the different transaction gas limits for each transaction.

On the other hand, Benji Bananas' developers should include pre-checks in these functions, calculating the estimated number of computations from the looping variable and update of contract state variables, against the current supply of gas left in the current block. They will be able to obtain a confidence interval where the transaction will most likely succeed. Otherwise, if the transaction has a high chance of failure, the transaction could be reverted with minimal gas burned from the computations, saving both the blockchain's and the users' resources.

3.5 Era7: Game of Truth

Era7 is a metaverse dApp game that is developed on the Binance Smart Chain (BNB Chain). It is a novel card-trading game that incorporates both fighting and strategy, mainly consisting of consecutive three-minute games. Players can mix and match their playing cards from their card library to obtain strategic combinations to win their opponents. Players can also obtain valued playing cards through collecting, fighting, trading, summoning, or synthesising certain cards. In addition, players can also profit from dividends or by sharing the game with friends.

Smart Contracts Overview

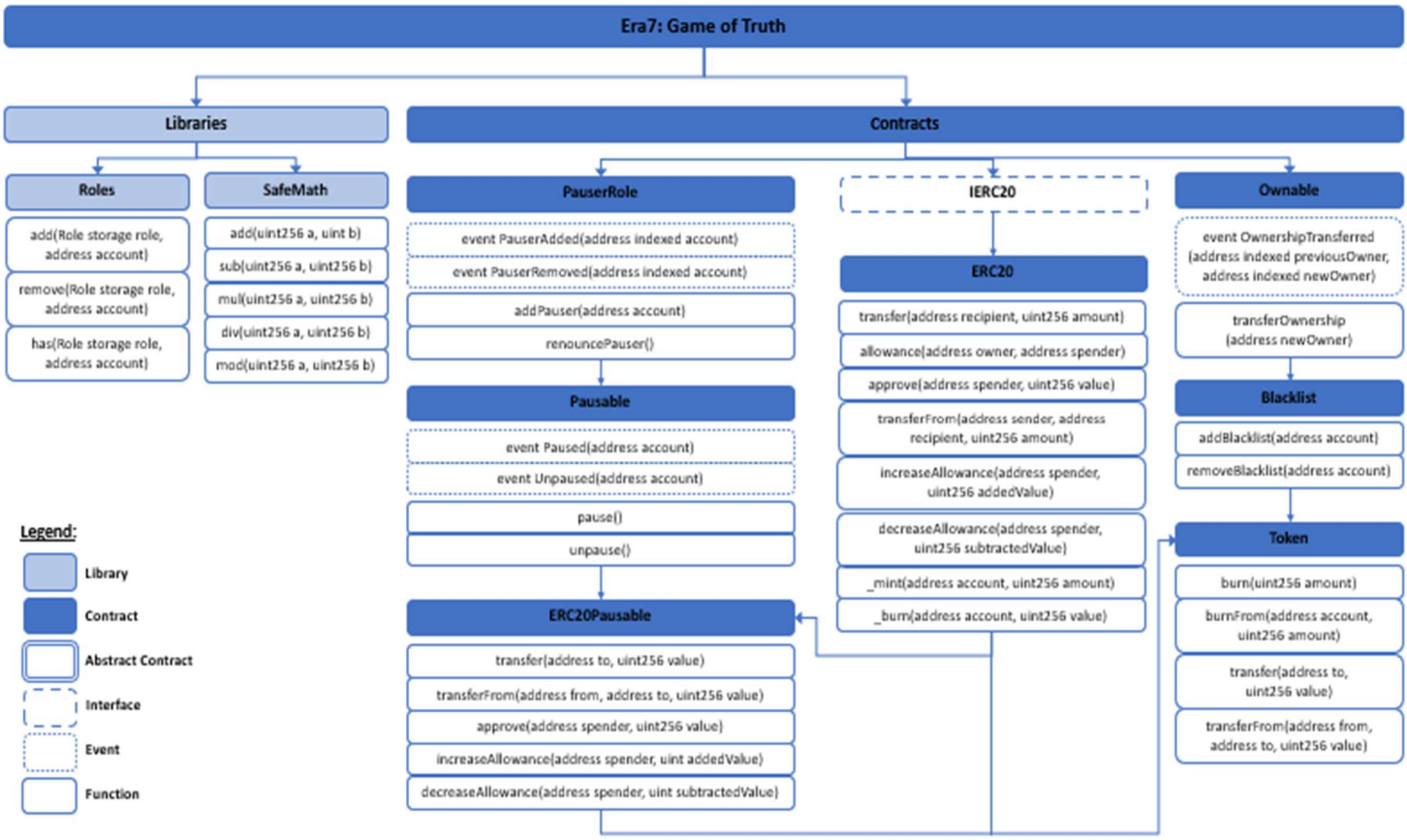

Figure 26: Overview of Era7's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Era7: Game of Truth	Satisfies	Satisfies	Violates	Satisfies

Figure 27: Summary of Era7 Fairness Analysis

Timing Standard Violation:

The _transferOwnership() function in the Ownable contract emits the event of OwnershipTransferred before the execution of ownership transfer, creating a chance for adversaries to launch a front-running attack, just like in Alien Worlds' Owned contract.

```
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Figure 28: _transferOwnership Source Code

```
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    originalOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(originalOwner, newOwner);
}
```

Figure 29: _transferOwnership Corrected Code

Using the same possible adversary attack example by Annie as in Alien Worlds' scenario, the developers should swap the 2 lines of code highlighted above, to ensure related transactions are triggered accordingly.

Additionally, Era7 also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.6 Iskra

Iskra is a dApp gaming platform built on the Klaytn and Ethereum blockchains that connects gamers and game studios. It offers a Launchpad for gamers to explore new game projects and allows users to trade in-game tokens and other assets as non-fungible tokens outside the game. ISK is the native token used for all services in the Iskra ecosystem. It has three utilities, including being used to buy Pioneer non-fungible tokens, as a governance token for voting on initiatives, and as a key currency for transactions and fees within the platform.

Smart Contracts Overview

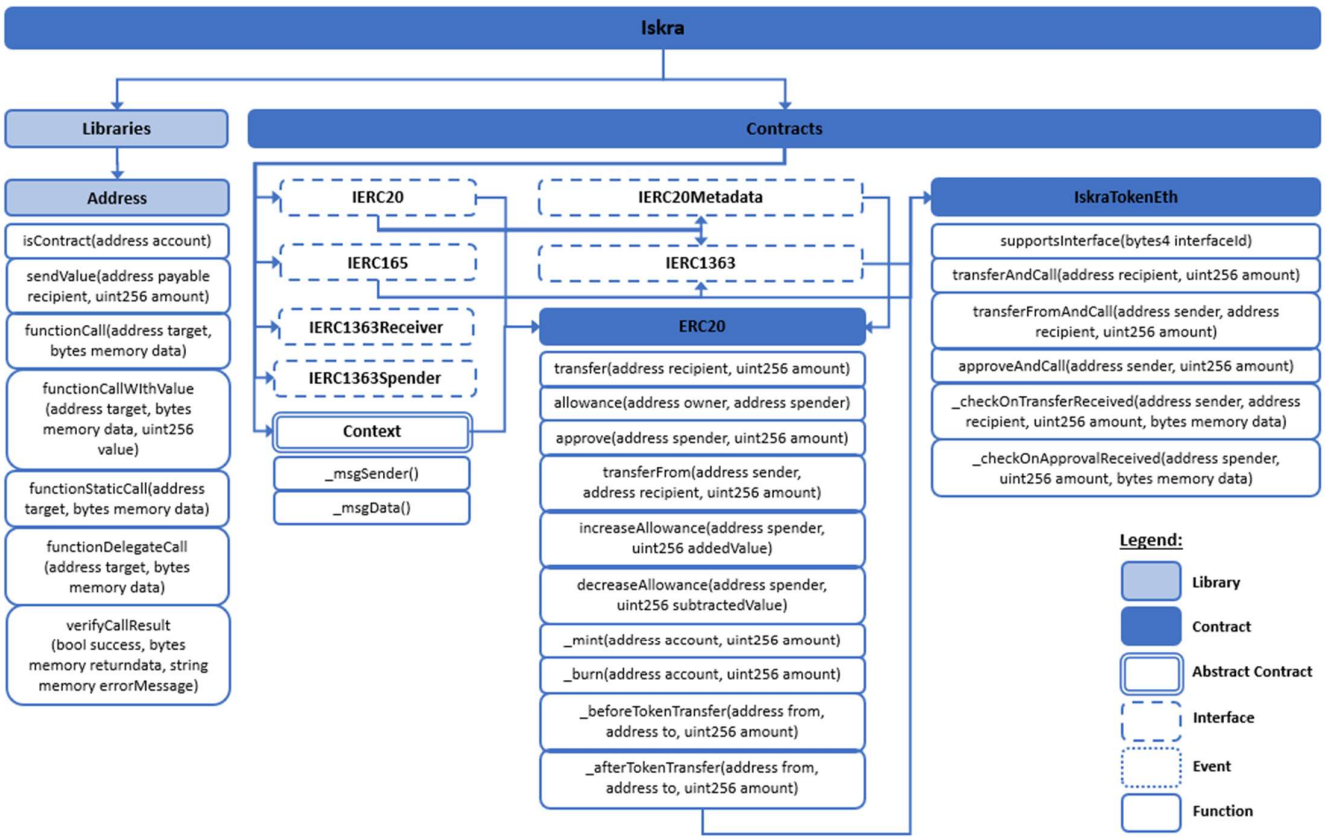

Figure 30: Overview of Iskra's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Iskra	Violates	Satisfies	Violates	Satisfies

Figure 31: Summary of Iskra Fairness Analysis

Execution Standard Violation:

The sendValue() function in the Address library passes the function call along with all remaining gas by default to the recipient without proper checks conducted upon the call's return. An adversary will be able to launch a re-entrancy attack on this contract.

```
function sendValue(address payable recipient, uint256 amount) internal {
    require(address(this).balance >= amount, "Address: insufficient balance");

    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Address: unable to send value, recipient may have reverted");
}
```

Figure 32: sendValue() Source Code

For example, the player Paul is sending value to the adversary Annie as recipient and executes a contract that make uses of this Address library and the sendValue() function. Paul's transaction passes the sufficient balance requirement check and passes the function call to Annie, along with the remaining gas. Annie now attains the control flow of the contract. Since Ether transfer can always include code execution, Annie could have set the recipient as a contract that calls back into sendValue() over multiple return calls before the first invocation of the function was returned, essentially retrieving all the Ether in the contract.

```
mapping (address => uint) userBalances;

function sendValue(address payable recipient, uint256 amount) internal returns (bool) {
    require(userBalances[msg.sender] >= amount, "Address: insufficient balance");

    uint bal = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Address: unable to send value, recipient may have reverted");
    userBalances[msg.sender] = bal - amount;

    return true;
}
```

Figure 33: sendValue() Corrected Code (Checks-Effects-Interactions)

There are several possible solutions to this re-entrancy issue. One possible solution is for the developers to extract the function out of the Address library into a contract implementation and use the Checks-Effects-Interactions pattern to design their code instead. Doing so will allow the contract to set the balance to 0 before passing the control flow to the recipient, preventing re-entrancy by the recipient. After the first invocation's return call, the transaction will reset the user's balance to the updated balance after deducting the transferred amount.

```
mapping (address => uint) userBalances;
bool private lockBalances;

function sendValue(address payable recipient, uint256 amount) internal returns (bool) {
    require(!lockBalances, "Balances are locked!");
    require(userBalances[msg.sender] >= amount, "Address: insufficient balance");
    lockBalances = true;

    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Address: unable to send value, recipient may have reverted");
    userBalances[msg.sender] -= amount;

    lockBalances = false;
    return true;
}
```

Figure 34: sendValue() Corrected Code (Mutex)

Another possible solution is to introduce a mutex which allows the contract to lock the current state where anyone else other than the owner will not be able to change any states in the contract. If other parties try to call sendValue() again before the first call is completed, the first require check for the mutex lock in the function will prevent any further actions by the call, solving the re-entrancy issue. However, when it comes to a multiple contract situation for the mutex implementation, careful design of the code is needed as adversaries can call functions to obtain the lock without ever releasing the lock, ultimately causing a deadlock, and stalling the transaction forever.

Timing Standard Violation:

Iskra also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.7 Meta Apes

Meta Apes is a mobile play-and-earn MMO strategy dApp game built on the Binance Smart Chain (BNB Chain). The game takes place in a post-apocalyptic world where humanity has become extinct, and with the world being ruled by apes. Players work together with their gangs to become the strongest clan and dominate space. Meta Apes uses a dual token model: SHELL which is the utility token and in-game currency, and PEEL which is the governance & gas token which all non-fungible tokens purchases are using to buy back and burn. Additionally, it is the first game to launch on BSC Application Sidechain (BAS), where the game will have their dedicated Ape Chain only used for the Meta Apes game.

Smart Contracts Overview

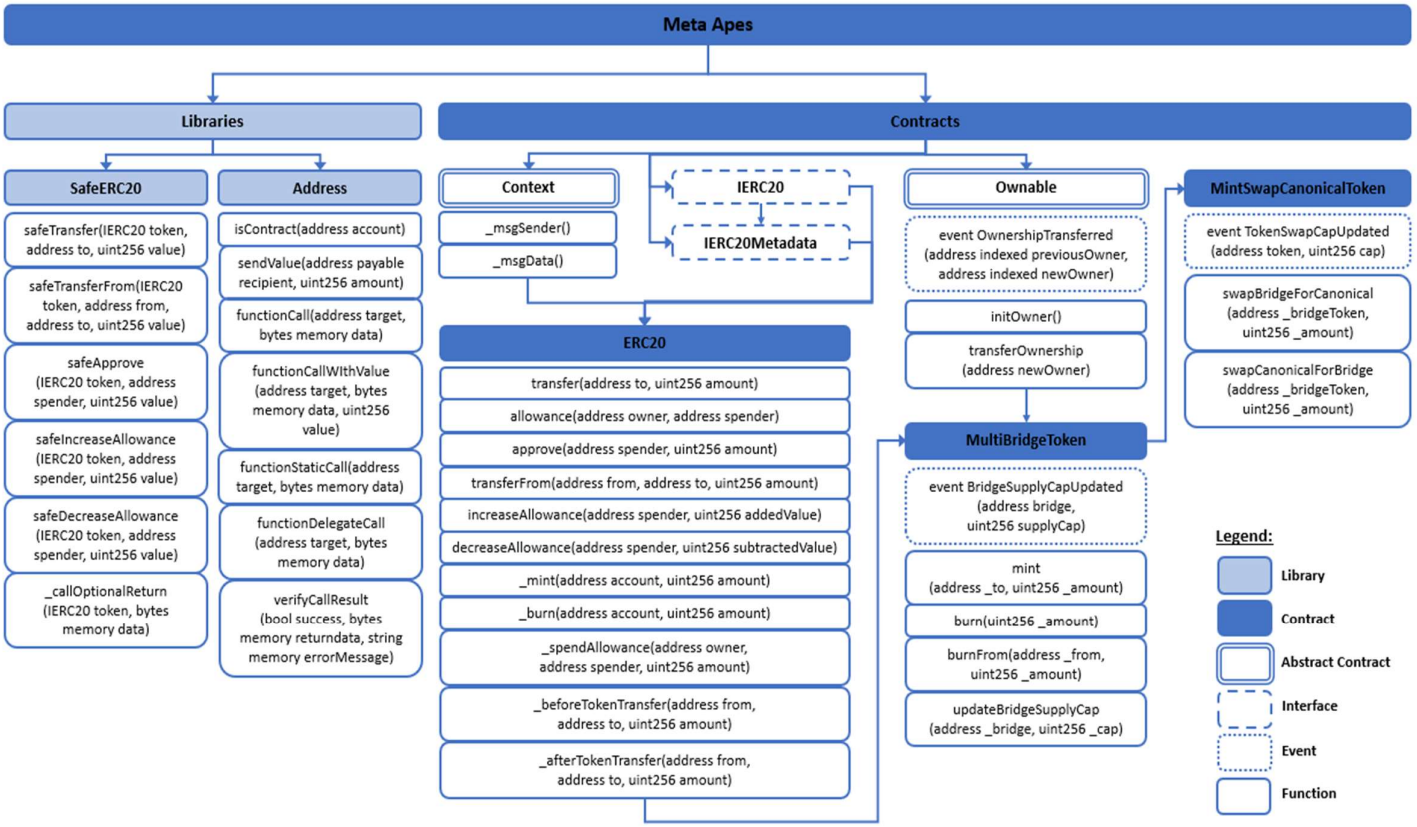

Figure 35: Overview of Meta Apes' Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Meta Apes	Satisfies	Satisfies	Violates	Satisfies

Figure 36: Summary of Meta Apes Fairness Analysis

Timing Standard Violation:

Meta Apes also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.8 Planet IX

Planet IX is a non-fungible token-based multiplayer strategy dApp game on the Polygon blockchain. The objective of the game is to restore a fallen planet through collecting, trading, and competing with unique digital creatures called PIX. The game map consists of 1.6 billion PIXs, which are individual non-fungible tokens that can be acquired using the game's own ERC-20 utility token called IX Token (IXT). These PIXs have different attributes and characteristics that make them valuable and rare, and they can be used to participate in battles and tournaments to win prizes and earn in-game currency. The game also includes a breeding system that allows players to create new PIXs with unique traits by combining two existing ones. As players progress in the game, they can form territories, develop technology, raid other players, and gain ownership of in-game corporations with different functions and perks.

Smart Contracts Overview

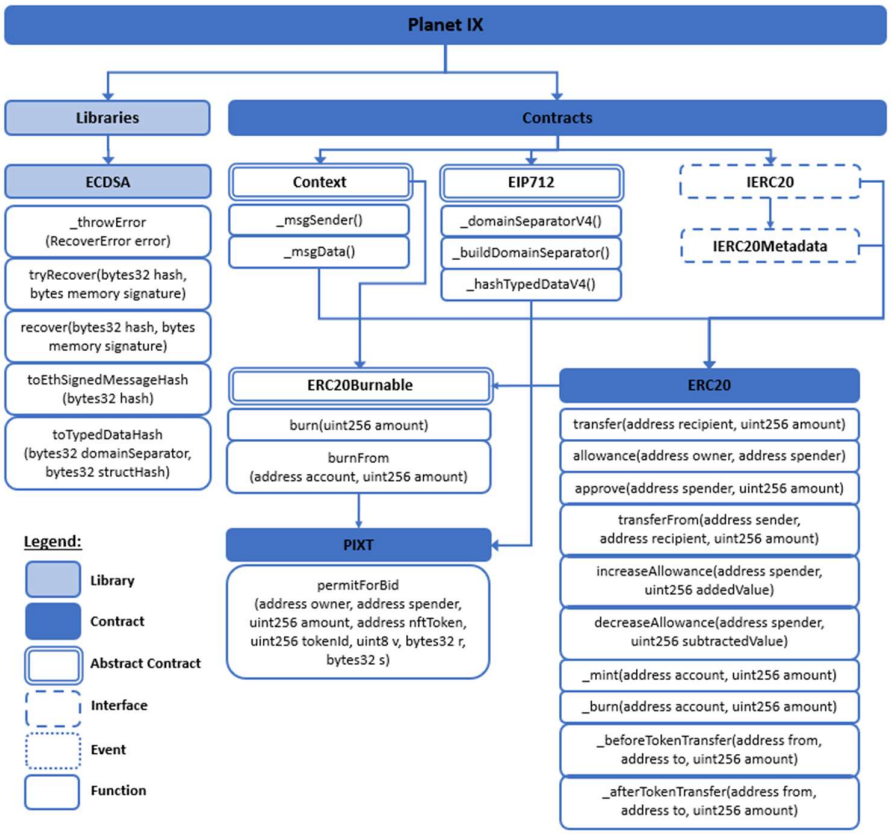

Figure 37: Overview of Planet IX's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Planet IX	Satisfies	Satisfies	Violates	Satisfies

Figure 38: Summary of Planet IX Fairness Analysis

Timing Standard Violation:

Planet IX also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.9 Playzap Games

Playzap Games is a community-driven gaming arena dApp built on the Binance Smart Chain (BNB Chain) that offer players a chance to win prizes and earn tokens while playing a variety of high-quality games. The platform includes multiple classic casual games such as Bingo, Solitaire, Bubble Shooter, and 8 Ball Pool, and offers real-time competition and frequent tournaments. The game currently features seven live games with daily competitions which players can win cryptocurrencies such as BUSD, $PZP, and other partner tokens based on their skill and loyalty.

Smart Contracts Overview

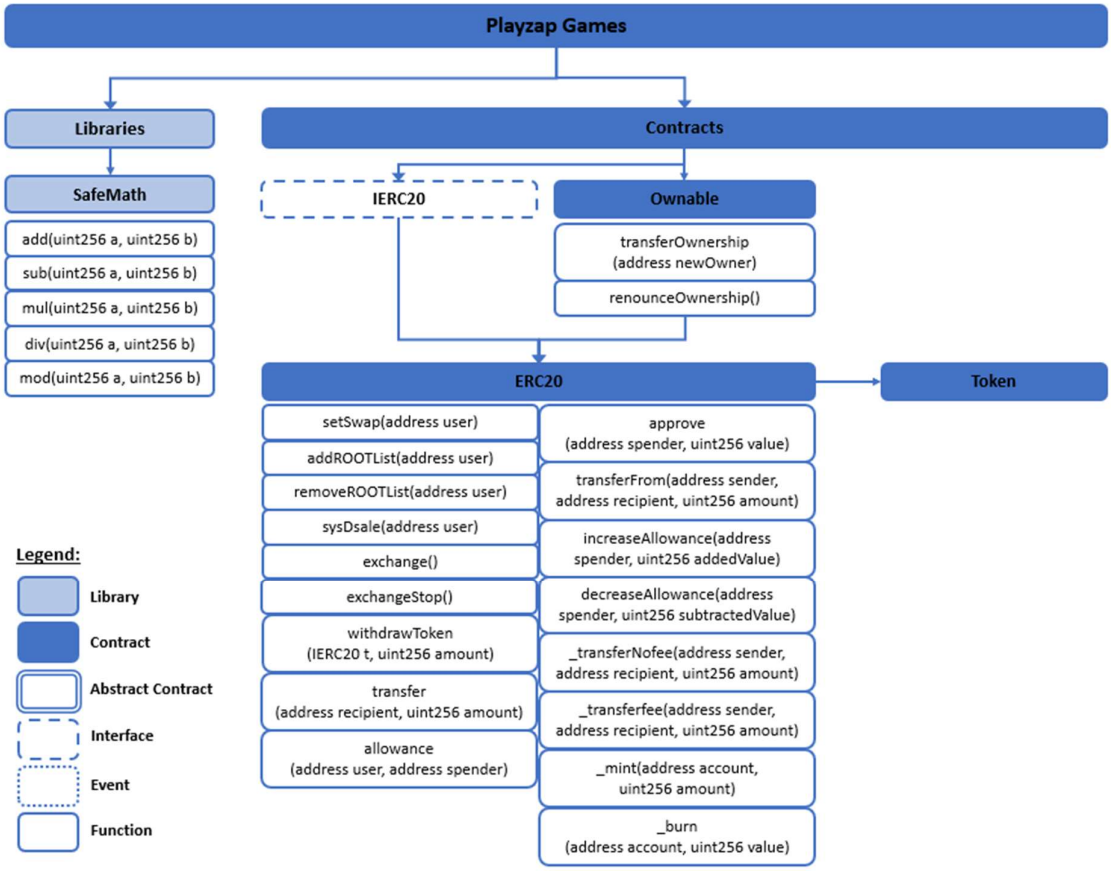

Figure 39: Overview of Playzap Games' Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Playzap Games	Satisfies	Satisfies	Violates	Satisfies

Figure 40: Summary of Playzap Games Fairness Analysis

Timing Standard Violation:

The renounceOwnership() and transferOwnership() functions in the Ownable contract are lacking either a return statement or an event emittance declaration to the block's event log, signalling the completion of the function call. Although this code design has low chances of being exploited by attackers, it may cause confusion or unexpected losses to the users of the blockchain.

```
function renounceOwnership() public onlyOwner {
    _owner = address(0);
}
```

```
function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _owner = newOwner;
}
```

Figure 41: renounceOwnership() and transferOwnership() Source Code

For example, Paul is transferring some tokens to Peter in the game with a few of Peter's contracts dependent on this transaction to begin executing. Upon transfer of the ownership, Peter finds that his contracts did not execute because there was no event log recorded that shows that he obtained the tokens, and his contracts were designed to be watching the event log. Ultimately, Peter will have to manually activate his contracts, missing the optimal timing of execution he intended and bearing the unexpected differences in token prices.

```
event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

function renounceOwnership() public onlyOwner {
    originalOwner = _owner;
    _owner = address(0);
    emit OwnershipTransferred(originalOwner, address(0));
}
```

```
function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    originalOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(originalOwner, newOwner);
}
```

Figure 42: renounceOwnership() and transferOwnership() Corrected Code

With the creation and usage of the OwnershipTransferred event, the blockchain is updated timely of the successful completion of these transactions and allows other dependent transactions to flow smoothly and as expected.

Additionally, Playzap Games also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.10 Sunflower Land

Sunflower Land is a dApp game on the Polygon Blockchain where the player is stranded on a deserted island and must gather resources to farm and grow their empire. Players can purchase and own virtual land in the game, and then create and customize their own unique virtual worlds on that land. The game includes thousands of unique tradeable in-game resources and rare Special Function Tokens (SFTs) that can be crafted each week. Sunflower Land uses non-fungible tokens to represent virtual land ownership, and players can buy, sell, and trade these non-fungible tokens on various cryptocurrency exchanges. The game also features various social and economic features, allowing players to interact with each other and earn cryptocurrency rewards through various in-game activities.

Smart Contracts Overview

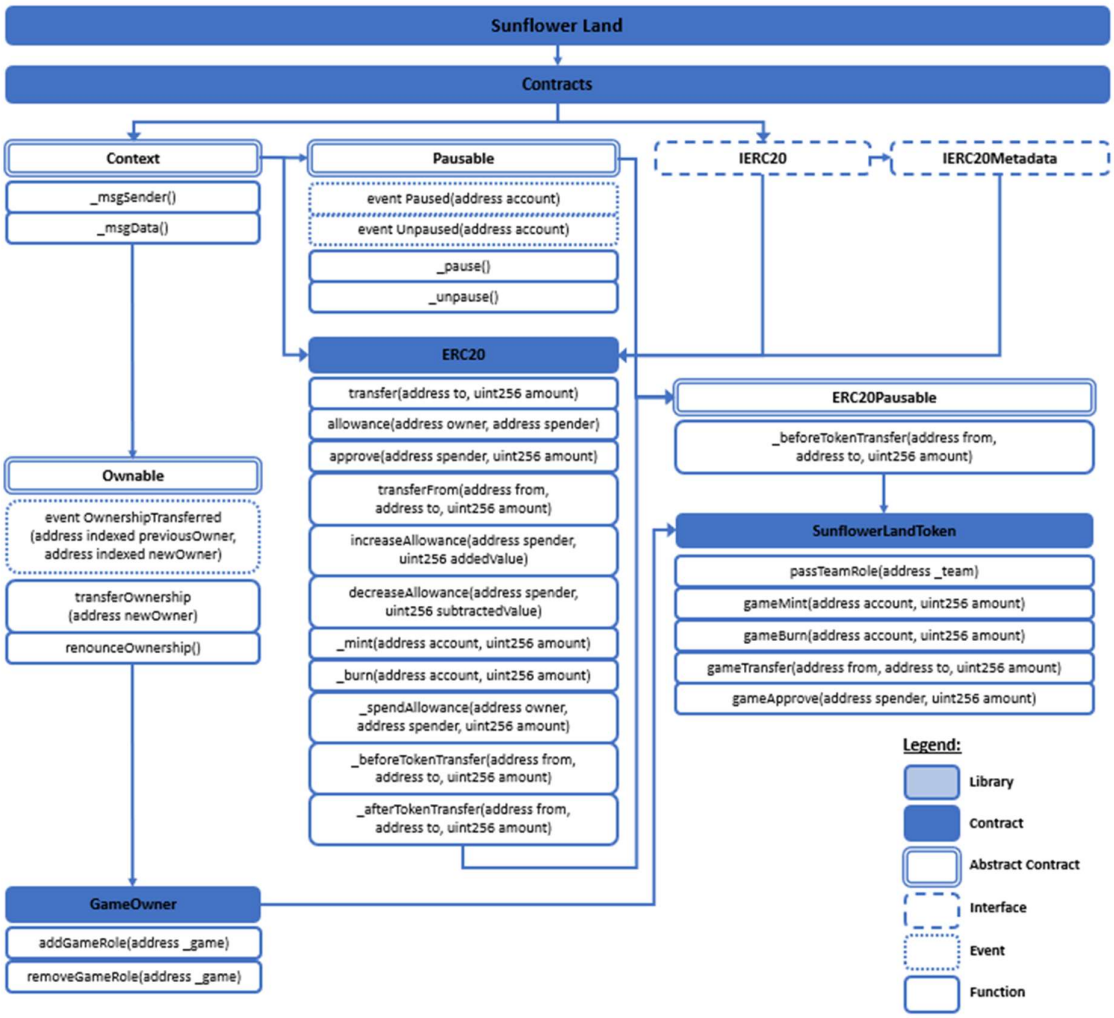

Figure 43: Overview of Sunflower Land's Smart Contracts

Analysis Results

dApp Game	Execution	Input	Timing	Integrity
Sunflower Land	Satisfies	Satisfies	Violates	Satisfies

Figure 44: Summary of Sunflower Land Fairness Analysis

Timing Standard Violation:

Sunflower Land also suffers from the same race condition violation for their approve() and transferFrom() functions in the ERC20 contract, just as explained in Arc8's analysis.

3.11 Results Summary

dApp Game	Execution	Input	Timing	Integrity
Alien Worlds	Violates *(Denial-of-Service)*	Satisfies	Violates *(Front-running)*	Satisfies
Arc8 by GAMEE	Satisfies	Violates *(Integer Overflow/Underflow)*	Violates *(Front-running)*	Satisfies
Axie Infinity	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies
Benji Bananas	Violates *(Logic bug)*	Satisfies	Satisfies	Satisfies
Era7: Game of Truth	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies
Iskra	Violates *(Re-entrancy)*	Satisfies	Violates *(Front-running)*	Satisfies
Meta Apes	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies
Planet IX	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies
Playzap Games	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies
Sunflower Land	Satisfies	Satisfies	Violates *(Front-running)*	Satisfies

Figure 45: Summary of Fairness Analysis

For execution standard, most of the dApps had a generally logical source code, except for the 3 violations stated. They were only apparent after scrutinising the source code and were unlikely to be experienced by real users of the dApp games. Thus, these are attacks that the developers should have been accounted for during development itself.

For input standard, with the aid of in-built arithmetic checking in newer Solidity versions and the SafeMath library for older versions, most of the dApps satisfied this fairness standard. The only exception being Arc8, where they used an older version without importing the SafeMath library and violated the standard due to integer overflow/underflow issue.

For timing standard, front-running attack vulnerability was the prominent cause for most of the flagged violations. Most of these were a resultant of using ERC20's functions that have a possible race condition situation, with the exception being Alien Worlds and Benji Bananas, who did not have the ERC20 contract. Similarly, these violations could have been prevented with thorough validation from the developers.

Lastly, for integrity standard, there were no violations as all the dApps contained well-designed source code with no possible security leaks. With all the chosen dApps being the top grossing dApps and are used daily, it was to be expected that the smart contracts will likely not contain phishing nor authentication keys in the source code.

Overall, there were no dApp that had satisfied all 4 fairness standards.

4 Conclusion

4.1 Conclusion

In this project, we had studied the common malicious attacks launched on the blockchains by attackers and used them as basis for 4 broad fairness standards. The standards were then applied on each of the 10 top grossing decentralised game applications to study if their smart contracts were fair to all parties in their transactions.

Based on our analysis results, there were no game that was flawless. Initially, this project shortlisted the top grossing dApp games after considering the trade-off between popularity and a fruitful analysis. Intuitively, more frequently played games should had been under larger scrutiny and received more revisions of source code to be more resilient to attacks. Although choosing random less popular dApp games will certainly have more standards violations and issues found, but the results will be less credible since these games will not be representative of the masses. Thus, after analysing the top games, the results obtained may suggest the possible hypothesis that there are currently no perfectly secured dApp game on the market.

4.2 Future Work

While the current implementation of our project meets the initial objectives, there are several areas for future work that could enhance the project's functionality and impact. These include:

- Setting up a systematic and quantitative evaluation framework: The current project utilises the expertise of the implementor to analyse smart contracts' source code, which is time-consuming and highly dependent on the implementor's knowledge and skill sets. Such workflow is also prone to human errors. There could be a defined automation involved that could eliminate the human aspect of the implementation and produce a more structured analysis.

- Expansion of fairness standards: With such rapid expansion of the decentralised world, the common malicious attacks evolve as well. The shortlisted fairness standards could be updated regularly using available online sources with statistics as evidence supporting the shortlisting of those standards. Also, with 4 being an arbitrary number of standards chosen for this project, there could be further analysis on the optimal number of fairness standards to be used for evaluation of the dApp games.
- Materialising the analysis results: After obtaining these analysis results with ample research and evidence, there could be follow ups with the dApp games' developers to correct those violations found. Ultimately, the analysis results were procured with the hopes of creating a safer blockchain community, eliminating these loopholes at their root causes will fulfil the motivation behind this project.

All in all, we believe that there is significant potential for this project to grow and evolve over time, and hopefully there will be more technical researchers who can continue the development and improvement of this project to provide a deeper view and understanding of the decentralised space.

References

A. Kosba, A. Miller, E. Shi, Z. Wen and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016, pp. 839-858, doi: 10.1109/SP.2016.55. Available online at: https://ieeexplore.ieee.org/abstract/document/7546538

Bscscan. (n.d.). *Binance (BNB) Blockchain Explorer*. Retrieved March 19, 2023, from https://bscscan.com/

Chaum, D. (1982). Advances in cryptology. SpringerLink. Available online at: https://link.springer.com/book/10.1007/978-1-4757-0602-4

Chaum, D. (1983). *Blind Signatures for Untraceable Payments. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds) Advances in Cryptology.* Springer, Boston, MA. Available online at: https://doi.org/10.1007/978-1-4757-0602-4_18

CoinMarketCap. (n.d.). *Global cryptocurrency market charts.* Retrieved March 20, 2023, from https://coinmarketcap.com/charts/

Dappradar. (n.d.). *Top blockchain games | dappradar*. Retrieved March 19, 2023, from https://dappradar.com/rankings/category/games

Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E. (2016). *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. Springer, Berlin, Heidelberg.* https://doi.org/10.1007/978-3-662-53357-4_6

Etherscan*. (n.d.). Ethereum (ETH) Blockchain Explorer*. Retrieved March 19, 2023, from https://etherscan.io/

GeeksforGeeks. (n.d.). *A computer science portal for geeks*. Retrieved March 20, 2023, from https://www.geeksforgeeks.org/

Github. (n.d.). *Shaojingle/FYP: Quantitative Analysis of Smart Contracts (Fairness)*. GitHub. Retrieved March 20, 2023, from https://github.com/shaojingle/FYP

Investopedia. (n.d.). *Investopedia*. Retrieved March 20, 2023, from https://www.investopedia.com/

Lam, E., & Kim, C. (2021, October 29). *Ether ($eth) rises to a record high as use of Ethereum blockchain surges*. Bloomberg.com. Retrieved March 20, 2023, from https://www.bloomberg.com/news/articles/2021-10-29/ether-rises-to-record-high-renewing-alt-season-expectations?sref=WD5fEjzY

Liu, Y., Li, Y., Lin, S., Zhao, R. (2020). *Towards Automated Verification of Smart Contract Fairness. Nanyang Technological University*

Loi, L., Duc-Hiep, C., Olickel, H., Saxena, P., Hobor, A. (2016). *Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.*

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system. Decentralized business review, 21260.* Available online at: https://assets.pubpub.org/d8wct41f/31611263538139.pdf

Polygonscan.com. (n.d.). *Polygon (Matic) Blockchain Explorer*. Polygon (MATIC) Blockchain Explorer. Retrieved March 20, 2023, from https://polygonscan.com/

Sigalos, M. K. (2022, July 1). *FBI adds 'Cryptoqueen' to Ten most wanted fugitives list after alleged $4 billion onecoin fraud*. CNBC. Retrieved March 20, 2023, from https://www.cnbc.com/2022/06/30/fbi-adds-cryptoqueen-to-ten-most-wanted-fugitives-list-for-fraud.html

Singh, A. (2019, October 22). *Blockchain Smart Contracts Formalization: Approaches and challenges to address vulnerabilities. Computers & Security.* Available online at: https://www.sciencedirect.com/science/article/pii/S0167404818310927

So, S., Lee, M., Park, J., Lee, H., Oh, H. (2020). *VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In 2020 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 825-841.*

Szabo, N. (1994). *Smart Contracts.*

Szabo, N. (1996). *Smart Contracts: Building Blocks for Digital Markets.* Available online at: http://www.alamut.com/subj/economics/nick_szabo/smartContracts.html

Szabo, N. (1997). *Formalizing and Securing Relationships on Public Networks. First Monday, 2(9).* Available online at: https://doi.org/10.5210/fm.v2i9.548

T. M. Hewa, Y. Hu, M. Liyanage, S. S. Kanhare and M. Ylianttila, *"Survey on Blockchain-Based Smart Contracts: Technical Aspects and Future Research,"* in *IEEE Access*, vol. 9, pp. 87643-87662, 2021, doi: 10.1109/ACCESS.2021.3068178. Available online at: https://ieeexplore.ieee.org/abstract/document/9383221

Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R. (2018). *An Overview of Smart Contract: Architecture, Applications, and Future Trends.*

Wood, G. (2014). *Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Project Yellow Paper 151 (2014), 1-32.*

Zhang, R., Xue, Rui., Liu, L., (2020, May 1). *Security and privacy on Blockchain*. ACM Computing Surveys. Retrieved March 20, 2023, from https://dl.acm.org/doi/10.1145/3316481