PM

Implementation of a P2P NFTs protocol (ERC-721) designed to mitigate ticketing industry hitches

MASTER'S DEGREE PROJECT

Francisco Luís Barros Pontes
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

January | 2023

Implementation of a P2P NFTs protocol (ERC-721) designed to mitigate ticketing industry hitches

MASTER'S DEGREE PROJECT

Francisco Luís Barros Pontes
MASTER IN INFORMATICS ENGINEERING

ORIENTATION
Karolina Baras

CO-ORIENTATION
Carlos Sérgio Figueira Faria

UNIVERSIDADE da MADEIRA

A Nossa Universidade

www.uma.pt

Faculdade de Ciências Exatas e da Engenharia

Mestrado em Engenharia Informática

Implementation of a P2P NFTs protocol (ERC-721) designed to mitigate ticketing industry hitches

Francisco Luís Barros Pontes

Orientado por:

Carlos Sérgio Figueira Faria

Professora Karolina Baras

Constituição do júri de provas públicas:

Professor Eduardo Miguel Dias Marques, Presidente

Professor Pedro Filipe Pereira Campos, Vogal

Wednesday, 29th March, 2023

Resumo

Os mercados secundários ilegais têm sido um incómodo de longa duração para a indústria da bilheteira. Os revendedores tendem a comprar agressivamente o maior número possível de bilhetes disponíveis, com o objectivo de os vender noutros locais por um lucro imenso. Após um esgotamento de bilhetes, os clientes interessados não têm outra escolha senão pagar a quantia exorbitante solicitada. Esta cadeia de eventos introduz um mercado injusto e desprotegido para os clientes em geral. Têm surgido propostas e implementações para diminuir esses efeitos, mas a solução perfeita ainda está para vir. Desenvolveu-se um protocolo sustentado pelo sistema Ethereum e que faz uso de NFTs com o objectivo de fornecer um sistema de bilheteira transparente, aberto, descentralizado, e honesto. Nesta tese incidiu-se fortemente na investigação em torno da segurança e técnicas de optimização para a escrita de smart-contracts. Além disso, nesta dissertação documentou-se detalhadamente o desenvolvimento, testes, e estratégias de validação utilizadas para a entrega do protocolo. Concluiu-se que embora o protocolo sirva notavelmente para o seu propósito, existe uma clara limitação, um mercado central autoritário. No entanto, acredita-se que o trabalho aqui implementado posssa servir de orientação para novos utilizadores destas tecnologias, ou servir de inspiração para ideias recém-nascidas baseadas em blokchain.

Keywords: Ticket scalping · Blockchain · Ethereum · NFTs · smart-contracts

Abstract

Illegal secondary markets have been a long-lived hassle for the ticketing industry. Scalpers tend to aggressively buy out as many available tickets as possible with the goal to sell them elsewhere for an immense profit. Upon a sold-out, interested customers have no choice than to pay the exorbitant amount asked. This event-chain introduces an unfair and unprotected market for the overall customers. Proposals and implementations to diminish those effects have been emerging, but the perfect solution is yet to come. We have developed a protocol underpinned by the Ethereum system which makes use of NFTs with the goal of providing a transparent, open, decentralized and honest ticketing system. This thesis strongly focused on research regarding security and optimization techniques for smart-contract coding. Moreover, this dissertation thoroughly documented the development, testing, and validation strategies used to deliver the protocol. We conclude that although the protocol notably serves its purpose, there is one clear limitation, a central authoritative marketplace. Nevertheless, we believe the work herein implemented can serve as guidance to onboard new users to these technologies, or serve as inspiration for newborn blockchain ideas.

Keywords: Ticket scalping · Blockchain · Ethereum · NFTs · smart-contracts

Agradecimentos

Agradeço em primeiro lugar à Yacooba e ao meu orientador, Carlos Faria, pela possibilidade de concretizar este projeto, bem como pela orientação e disponibilidade que me foi dada.

Agradeço também à professora Karolina Baras pela sua orientação e disponibilidade.

Agradeço à minha família, namorada, e amigos chegados pelo apoio, motivação, e conselhos que me facultaram ao longo deste período.

Table of Contents

List of Figures

List of Tables

Listings

List of acronyms

CE Centralized exchange

DAO Decentralized autonomous organization

EOA External owned account

ERC Ethereum request for comments

ETH Ether

EVM Ethereum's virtual machine

IBAN International Bank Account Number

NFT Non-fungible token

PoS Proof-of-stake

PoW Proof-of-work

RAM Random access memory

SPV Simplified payment verification

URI Unique resource identifier

UTxO Unspent Transaction Output

1 Introduction

Since the beginning of ticketing industry, there has been a struggle with ticket resale in secondary markets for external profit. In regards to this activity, two names used to refer to ticket resellers have originated - brokers and scalpers. The main difference is that brokers are licensed and work for a firm whereas scalpers typically act illegally [1]. Over the last few decades, the technological evolution provided the capability to buy tickets remotely, increasing the opportunities for scalpers to act. As an example, fees of up to US$500 were paid to scalpers to get an appointment for a visa interview at the German consulates in Beirut, Tehran, and Shangha [2]. Furthermore, scalpers often sold appointments at prestigious hospitals in China, which required online booking, for up to 50 times their original value [2]. A New York Times analysis stated that resellers were making $60 million per year on just the Hamilton show alone [3].

To circumvent situations like these, there is a need to adopt behaviors which can deliver a fair and transparent market to the end-user. This project was hand-picked from Yacooba's [1] - a local startup - roadmap. By mixing blockchain ticketing protocol and curated travel bundles, Yacooba generates extra revenues for event promoters and rewards for audiences. The project consists of developing the protocol for event tickets, allowing ticket owners to resell their assets according to event specific rules. There will be a major focus on efficiency and security, since the protocol will be open to everyone throughout a decentralized network.

In this paper, we document our development along with its backbone technologies. We start by detailing blockchain, a cryptographically secure and immutable type of database. Next, we introduce Ethereum, a decentralized and censorship resistant blockchain network with storage and computing capabilities. Later, we describe NFTs (Non-Fungible tokens), a novel concept coined on the Ethereum ecosystem for digital ownership. After introducing the underlying cornerstone technologies, we dive into the system's protocol. We explain each entity separately and how they all operate together. Two more components, the Platform and the Event check-in, are also detailed. The following chapter, Research, is the result of diligent research regarding smart contract security and optimization, blockchain scalability, and implementation of royalties over blockchain protocols. Then, the paper is enriched with relevant related work, referencing the solutions designed/proposed by other authors to mitigate ticket scalping. With research in place, we discuss the development, testing, and validation phases, where implementation progress was documented, as well as any bottlenecks and overcomes.

1.1 Blockchain

Blockchain at its most basic concept can be seen as a cryptographically secure immutable database, which means that a digital signature is required to append new data [4] [5]. The data is spread throughout a sequential chain of blocks, where each block holds a number of transactions, akin to a trivial ledger. The maximum number of transactions that a block can contain depends on the block size and the size of each transaction. Users interact with the blockchain using a generated address, which means no user private information is stored on-chain. This provides privacy and anonymity [6].

[1] https://yacooba.com/

1.1.1 Architecture

Figure 1 represents the blockchain architecture. The first block is called the genesis block. A block is separated in two divisions, the block header and the block body. The header includes the block version, the parent block's hash, the Merkle tree root hash, the timestamp, the nBits and the nonce. On the other hand, the body contains a transaction counter and all the transactions.

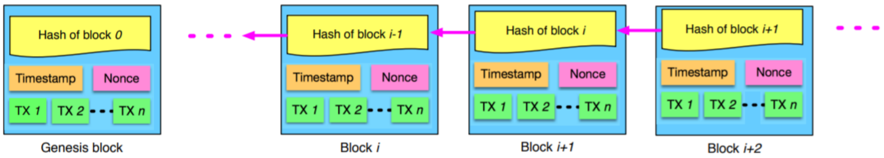


Fig. 1: Blockchain Architecture [6]

1.1.2 Digital signature

Besides making use of hash cryptography, blockchain typically utilizes elliptic curve cryptography to generate the public-private key pair associated with a user's wallet [6]. The public key can be generated from the private key, but not the other way around - the math only works one way [7]. To better understand the double key concept, an analogy can be made. If we consider our bank account, the IBAN is to the public key what the card PIN is to the private key, being the latter the one used to digitally sign transactions.

The digital signature is a two-step process where transaction signing and verification take place.

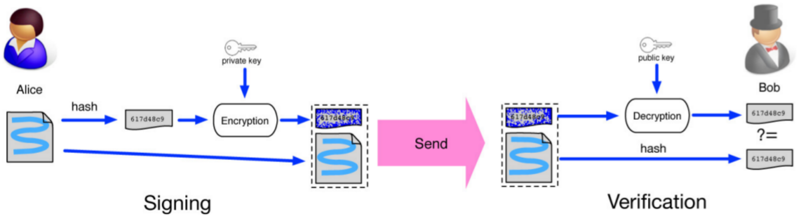


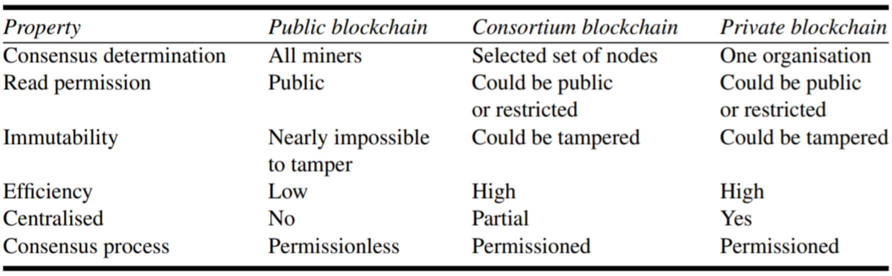
Fig. 2: Digital signature explanation [6]

Figure 2 shows how the digital signature is processed. A user starts by hashing the transaction data with his private key and sends it to a different recipient. Later, the receiver decrypts the data by using the sender's public key and verifies if the hashes match. This verification process is performed by the network nodes.

1.1.3 Types of blockchain

Due to a variety of blockchains with different characteristics, we need qualifiers to fully understand the type of blockchain we are interacting with. Mainly, we need to realize if the given blockchain is open, public, global, decentralized, neutral, and censorship-resistant [7].

Nevertheless, current blockchain systems can be identified as public, private or consortium [6] [8]. Public ones are often called permissionless, whereas private or consortium blockchains are known as permissioned. A permissioned blockchain can be seen as a corporate controlled intranet, being the permissionless like the public internet, where anyone can participate. Permissioned blockchains are often deployed for a group of organizations and individuals, typically referred to as a consortium [4] [9].



Property	*Public blockchain*	*Consortium blockchain*	*Private blockchain*
Consensus determination	All miners	Selected set of nodes	One organisation
Read permission	Public	Could be public or restricted	Could be public or restricted
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralised	No	Partial	Yes
Consensus process	Permissionless	Permissioned	Permissioned

Fig. 3: Types of blockchain [6]

Figure 3 sums up the differences between the supra-mentioned blockchain categories. As expected, consortium and private blockchains are rather centralized and a group of participants have management capabilities over the data and network visibility. On the contrary, public blockchains are decentralized and accessible by anyone. The decentralization comes at an efficiency cost.

1.1.4 Finality

Finality is the certainty that blocks will not be reverted once appended to the blockchain [10]. The blocks could be reverted due to a fork on the chain. Here we cover two types of finality: probabilistic and absolute. Probabilistic finality means that the probability that a block is not reverted increases as the blockchain grows, i.e., as more blocks are added. On the contrary, absolute finality indicates that transactions are instantly finalized once added to the blockchain [11]. Finality is an important concept to keep in mind when dealing with blockchains.

1.1.5 Consensus

To achieve network security, blockchains utilize a consensus mechanism. These may vary, as the blockchains also vary. Over the following paragraphs, we explain the most common ones, proof-of-work and proof-of-stake.

Proof-of-work (PoW in short) is a consensus model that rewards participants for their work. These participants, often called miners, use their hardware to solve a cryptographic puzzle where the answer is the hash value of the block which meets the network requirements. The miner who first solves the puzzle has the right to create a new block [6] [11]. Once a miner solves the puzzle, he transmits his block with a valid nonce to full nodes (the nodes hosting the network) in the blockchain network. Those nodes verify that the new block fulfills the puzzle requirement, add the block to their copy of the blockchain and emit the block to their peer nodes. For that reason, the new block is rapidly broadcast throughout the network. Verification of the nonce is easy since only a single hash needs to be computed to check if it solves the puzzle [4]. PoW belongs to the probabilistic-finality consensus protocols since it guarantees eventual consistency. This consensus mechanism is the one used by a majority of the digital cryptocurrencies, such as Bitcoin, Litecoin, and Dogecoin. [12]

Proof-of-stake (PoS in short) is an energy-saving alternative to Proof-of-work. It relies on the belief that people with more staked currency are less prone to attack the network. Although publishing nodes still need to solve a puzzle, there is no need to try multiple different nonces. Instead, the key for the puzzle is the amount of stake [11]. The methods which the blockchain network uses for determining the new block publisher may vary between random selection of staked users, multi-round voting, coin aging systems and delegate systems. Regardless of the exact approach, users with more stake are more likely to publish new blocks [4]. One well-known cryptocurrency project that makes use of PoS is Ethereum. Curiously, Ethereum currently uses both PoW and PoS, having two distinct networks. Further, a merge to pure PoS is imminent [13].

1.2 Ethereum

In 2013, Vitalik Buterin proposed Ethereum, a model that would extend the possibilites of Bitcoin [7]. Ethereum's purpose is to embed the capability a computer has to store and execute programs within a decentralized blockchain, hence creating a distributed single-state (singleton) world computer. Ethereum's programs assemble a common state secured by the underlying consensus mechanism. Ether (ETH) is the currency used in the system to pay for the execution of programs and to limit the resources used by them, thereafter allowing for a sustainable Turing-complete computation [7]. Ethereum allows developers to produce economically oriented decentralized applications (DApps). It delivers high availability, auditability, transparency, and neutrality. In addition, it reduces or eliminates censorship and diminishes certain counterparty risks [7]. The stored programs are known as "smart contracts", typically written in high-level programming languages like Solidity, later being compiled to bytecode and executed on a virtual machine (Ethereum's Virtual Machine - EVM). Smart contracts are immutable deterministic scripts stored on-chain identified by a unique address. They run on the EVM whenever a transaction calls them. [14]. When interacting with a smart contract from a transaction, funds can be sent to them or externally exposed methods can be invoked. Failed transactions will be rolled-back, being registered as having been attempted. Albeit having failed, the Ether used to pay for gas costs of these transactions will be consumed. Gas - bought with ETH before transaction execution - is the mechanism employed by Ethereum to limit the resources that any program can consume [7]. Each executed instruction of the script adds to the final gas cost to be paid.

1.2.1 UTxO versus account based transaction models

The cryptocurrency protocols have unique characteristics based on the transaction models they are built on. The predominant ones are UTxO (e.g. used by Bitcoin) and account based (e.g. used by Ethereum) [15], even though new derivatives have been emerging based on them (e.g. used by Cardano [16]). On the next few lines, we sit both models face to face, to have a rougher understanding of differences between Bitcoin and Ethereum.

The UTxO (Unspent Transaction Output) model does not rely on accounts nor balances. Instead, the user has funds left from each transaction. For a neat analogy, we can think of going to a store and buying a 5 euro item with a 10 euro bill. In this situation, we will give all the 10 euros and, in return, the cashier will provide us with a new output of 5 euros. With this model, the total balance of a user is the compound of all UTxOs. This concept is designed to provide extra privacy, as the receiving address can be unique for each UTxO. Moreover, due to the stateless nature of the UTxOs, transactions can't affect the same output. For the same reason, there is no chance for a transaction to be replayed, which allows for more scalability. As for the downsides, we have limited capabilities to program smart contracts, since no state can be applied and the UTxOs have intrinsic spending criteria. [17]

The account based model is akin to a bank account. Given the account number, we can get the balance. This implementation gives us more flexibility to develop smart contracts, as business logic based on a persistent global state can be put in place. In addition, transactions become smaller than on the UTxO model because they do not need to have the final state. With more flexibility, there is generally more complexity, and that is one of the major drawbacks of this solution. Parallel executions need to be handled carefully due to the existence of state. Lastly, it is a less privacy preserving concept than the UTxO model, considering the transactions are linked to a single address. [17]

1.2.2 Ethereum's blockchain data structures

On Merkle trees, a hash oriented version of Binary trees [18], every node is the result of a hashing computation and, ultimately, the root hash is based on all hashes below it. Hoewever, these trees are inefficient thus Ethereum uses an optimized version of them, Patricia Merkle Trees [19]. These trees are typically aliased as tries, a tree data structure used for locating specific keys from within a set [2], a label we use going forward.

Revisiting figure 1, we learnt that a block is comprised of the header and the body. On the Ethereum blockchain, the header of the block contains, among other properties, the root value of three tries: transactions, state, and receipts. Each block instantiates and is directly associated with two new ties: receipts and transactions. The state trie, conversely, is a singleton and is continuously updated. Each entry in the state trie is associated with a storage trie - one for each account - through the root's hash. These connections are reflected on Figure 4.

Diving deeper into the relationship between the state and storage tries, we expose Figure 5. It can be observed that each entry is comprised of a key value pair, where the key is the address of an account and the value a combined encoding of the nonce, balance, storageRoot, and codeHash (for contracts). Each account, either EOA (External Owned account) or a contract, has a unique entry on the state trie. Every account is associated with a storage trie, connected via the storageRoot property.

[2] https://en.wikipedia.org/wiki/Trie

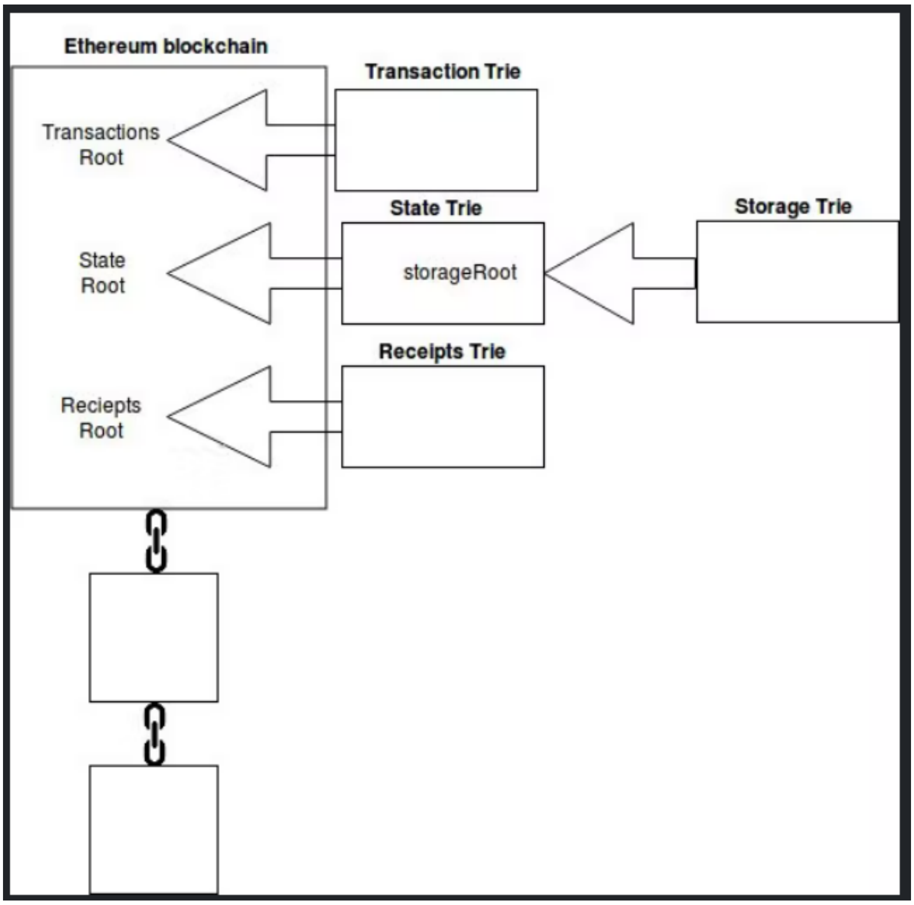


Fig. 4: High level view of tries on the Ethereum blockchain [20]

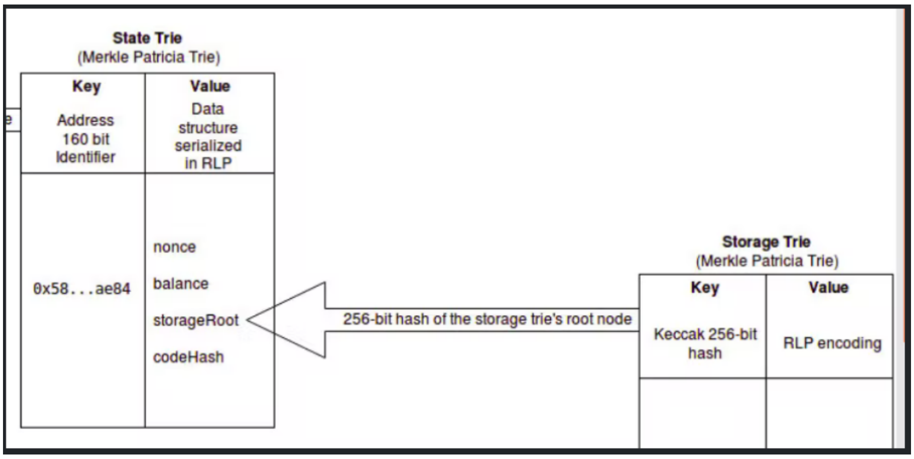


Fig. 5: Relationship between state and storage tries [20]

As a final insight, at figure 6 we peruse the relationship between the block and the transaction tries. As we know, each block will have its transaction trie. Each transaction on the trie represents a combined encoded value of the nonce, gasLimit, gasPrice, and value.

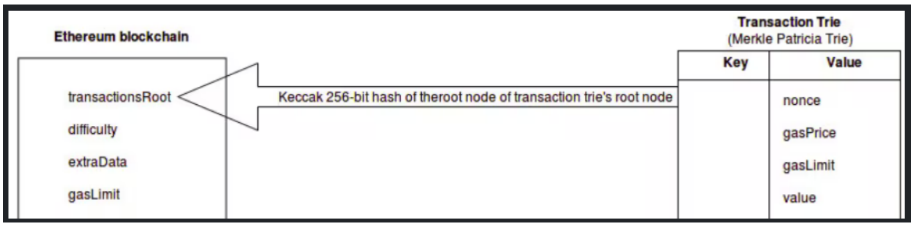


Fig. 6: Relationship between block and transaction tries [20]

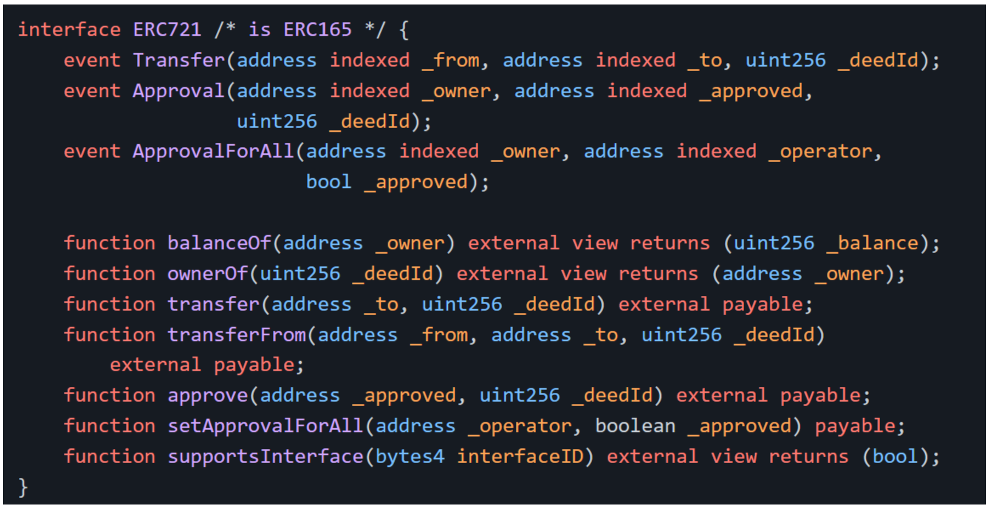
1.2.3 EVM's storage structures

The EVM is underpinned by a stack which uses 256-bit sized items and has a maximum depth of 1024 positions. Moreover, the machine is packed with a memory structure, volatile, and a permanent storage structure which is intrinsic to the system's state [21].

1.3 NFTs

NFT stands for Non-Fungible Token, a unique and not interchangeable token. These tokens can be utilized to digitally represent ownership of an asset. A standard originated on Ethereum to represent these type of tokens, the ERC721 (Ethereum Request for Comments 721) [22] . This standard defines a common way to identify and implement tokens of this type. In its core, the NFT is held by an owner and can be transferred to other addresses. Due to the openness and transparency intrinsic on the Ethereum blockchain, anyone can have a look and see who owns a specific NFT, along with the history of previous owners.

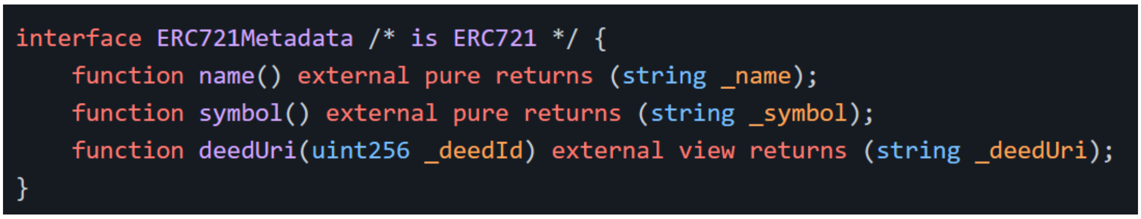
Figure 7 represents the mandatory interface of the ERC721 standard. All those events and functions need to be implemented in the smart contract. An event is the way used to log information. Having self-explanatory function names, this interface reveals methods to query ownership of a token, to transfer the token, or to give transfer authorization to third-parties.



```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
    event Approval(address indexed _owner, address indexed _approved,
                   uint256 _deedId);
    event ApprovalForAll(address indexed _owner, address indexed _operator,
                         bool _approved);

    function balanceOf(address _owner) external view returns (uint256 _balance);
    function ownerOf(uint256 _deedId) external view returns (address _owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId)
        external payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operator, boolean _approved) payable;
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

Fig. 7: ERC721 interface [7]

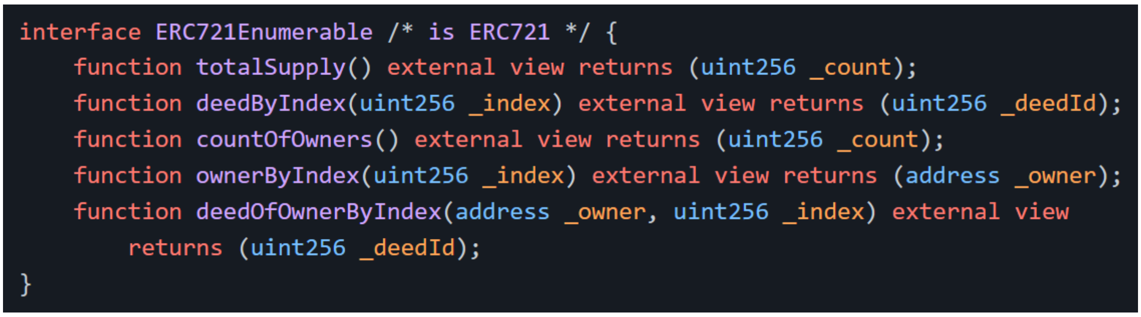
Extending the core functionality of NFTs, there are two broadly used interfaces, the ERC721Metadata and the ERC721Enumerable, which are represented on Figures 8 and 9, respectively.



```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string _deedUri);
}
```

Fig. 8: ERC721 metadata's interface [7]

The ERC721Metadata interface allows the attachment of characteristics to the collection of tokens. If this interface is applied, the collection has to have a name and a symbol, and each member of the collection has to have a URI (Unique Resource Identifier). The URI is what associates the token with a media resource (e.g. image).



```
interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256 _deedId);
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address _owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external view
        returns (uint256 _deedId);
}
```

Fig. 9: ERC721 enumerable's interface [7]

The ERC721Enumerable interface insights methods which can be useful for statistics, such as the total supply of token and the count of owners.

1.3.1 Use cases

From our perspective, NFTs can be employed wherever there is a need to prove ownership. So far, they have had greater impact on gaming, art, event, and collectibles [23].

On gaming, they really have the potential to disrupt and streamline trading of in-game assets [24]. NFTs would abolish the need to utilize secondary offline markets to sell them. Further, they establish the opportunity for interoperability of game assets. An asset bought on game A could be reused on game B. Similarly, termination of a game would not mean you lose the asset you paid for. Gracefully, the item will still live on the blockchain and be owned by you.

Regarding the art industry, there is an opportunity for artists to receive royalties for every trade on their creations. Copyrights are assured. It empowers artists with an innovative and seamless way to expose and trade their art [24].

1.4 Document structure

The introduction is followed by an explanation of the concept of the project, in chapter 2. Sequentially, chapter 3 is focused on research and considers security and optimization patterns, approaches to scale blockchains, and how royalties can be handled on blockchain protocols. Chapter 4 explores related proposals with different techniques to handle ticket scalping. Later, chapter 5 documents business requirements and the development which took progress to achieve them. Chapter 6 covers the testing and validation strategies used on the project. Ultimately, in chapter 7, the conclusion effectively summarizes the dissertation, denoting achievements as well as limitations.

2 Concept

This project is part of Yacooba's roadmap. By mixing blockchain ticketing protocol and curated travel bundles, Yacooba generates extra revenues for event promoters and rewards for audiences. The proposed project consists of developing and improving the on-going Yacooba's protocol which envisions a transparent, honest, secure, and decentralized ticketing system. In this section, we detail the general concept which hopes to mitigate ticket scalping. Further information of what in fact was developed/enhanced during this thesis can be found on the document's Development section.

2.1 Protocol

The protocol will live on the blockchain and autonomously serve arbitrary users who wish to:

- create events
- configure existing events
- buy tickets for an event
- resell own tickets

It mainly targets two types of actors, the event producer, and the final consumer - the event attendee. Besides that, it allows Yacooba administrators to configure a new admin account, change the protocol's fee, and pause/resume the contract. The protocol, represented on Figure 10, is comprised of 5 components: the forwarder; the factory, to generate new events; the event; the royalty database; and finally, the marketplace. We will move on to examine each of them.

2.1.1 Forwarder

Typically, users would have to pay ETH for transaction gas costs when interacting with the Ethereum blockchain. However, the community has introduced a proposal for meta-transactions, where user A creates and signs the data off-chain, but it is user B who triggers executions and pays for the transaction [25]. This is specially useful when the goal is abstract complexity from the end-users. In our protocol - Figure 10 - the entry point is the forwarder contract, which is responsible for relaying the transaction to the recipient contract, while providing additional useful metadata such as the transaction's signer's address.

2.1.2 Factory

The factory, as it suggests, produces and deploys new events. A fee is paid if the event is not created from within Yacooba's platform. To generate new events, it utilizes a proxy implementation from Open Zeppelin. [3] With this approach, deployment costs are drastically reduced, as the new deployment, the proxy, is a minimal contract which delegates transactions to the contract with the event implementation. In this setup, the event implementation executes the transactions within the context of the proxy contract thus any storage changes also apply to the latter.

2.1.3 Event

The event structure consists of 5 contracts. At the time of deployment, only the Event contract is considered since it inherits all the other ones.

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/Clones.sol

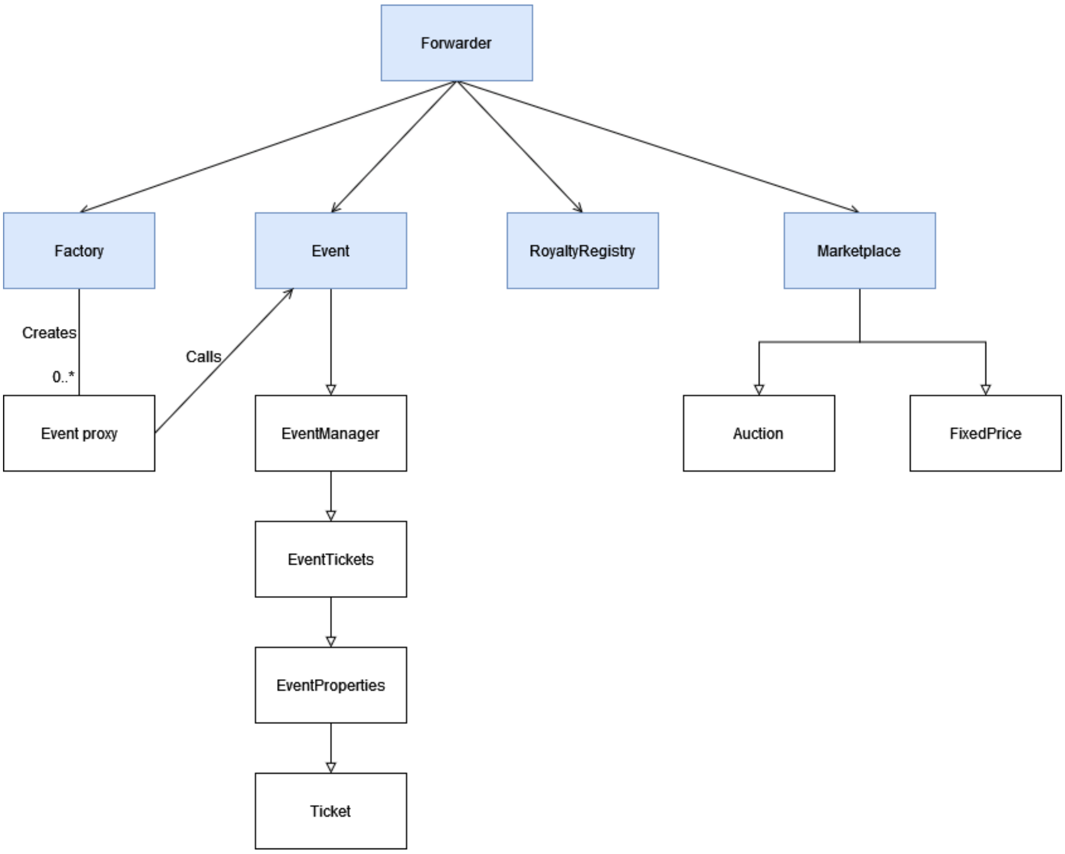


Fig. 10: Protocol's architecture

- Event - through heritage, contains all variables and methods from the other contracts. Moreover, implements a constructor routine to properly initialize a new event.
- EventManager - provides event-specific management capabilities to authorized users. For the event producer role, permits changing ticket supply; event dates; adding or disabling ticket tiers; withdrawing funds; setting the marketplace margin; and controlling the marketplace availability. Further, the protocol admin can change the protocol's fee; modify the protocol's admin's address; and pause/resume the contract.
- EventProperties - provides all event's properties' getters and setters. Properties include: event dates; protocol's fee; marketplace's address, margin, and availability; addresses of accepted payment token, protocol's admin, and royalty registry; the hashed access control roles.
- EventTickets - transfer-focused, possesses functions to buy - aka mint - new tickets and transfer them. Transfer validations occur therein. Moreover, stores ticket and ticket tier information.
- Ticket - contains the NFT definition inherited from the ERC721 standard

2.1.4 RoyaltyRegistry

This contract is intended to work as a separate repository which maintains all royalty related metadata, such as the royalty's receiver, percentage value, and minimum fixed value.

2.1.5 Marketplace

The marketplace is comprised of two contracts, Auction and FixedPrice. As the naming implies, Auction allows actors to bid on a given asset. On demand, the auction is closed and the asset transfered to the highest bidder. On the other hand, FixePrice empowers users to list items to be sold or buy listed items at the defined price.

2.2 Platform

Yacooba has been continuously developing its platform to provide a fully-functional and good looking user interface for event attendees and producers. The platform abstracts all the underlying blockchain complexity while still allowing web3 savvy users to utilize web3 providers for interacting with the blockchain. In addition, it is technically possible to perform direct contact with the living protocol, as long as the contracts' addresses are publicly known.

For the event check-in, Yacooba has conceived an interface on which event producers can scan and validate attendees' QR codes. The check-in is a moment of extreme importance when the goal is mitigate ticket scalping which is why we explain it in more detail in the following section.

2.3 Event check-in

Arriving at the event, the attendee will be asked to show the QR code. If the attendee registered on the platform using a web3 provider, he will have to use it to sign a message to generate the QR code's payload. Otherwise, the QR code will be available on the platform. Independent of the option performed, the account's private key is used for the message signing. The code is only valid for 5 minutes and will be validated against a list of attendees and a list of tickets already used. Authenticity is powered by a digital signature, the encryption model explained on figure 2.

3 Background

Within this chapter, we provide knowledge yielded from research with the goal to technically prepare us for the implementation phase. As most of the work pertains to writing smart contracts in Solidity, a deep-dive on security and optimizations best practices was put in place. Further, we describe how layer 2 solutions are being built to improve limitations of the Ethereum's blockchain. Finally, investigation on strategies to implement royalties for blockchain protocols was realized.

By leveraging contract-oriented programming languages, such as Solidity, developers can code smart contracts. These deterministically operate on the network nodes, upon demand, ending up changing the blockchain state. They serve as a ledger for users' assets, and, once deployed, they are immutable [26] [27].

3.1 Security

Throughout software development, security is an important aspect to pay attention to. When writing smart contracts, worrying about security is even more fundamental. This is due to the fact smart contracts are public and immutable once deployed, meaning anyone can possibly find unfixable bugs! Furthermore, smart contracts are typically used to exchange and hold users' assets, which means any exploit could incur big financial losses.

Considering the importance of writing secure smart contracts, we have investigated common exploits along with best practices to avoid them. Moreover, we've identified code analysis tools which aim to identify and fix these vulnerabilities.

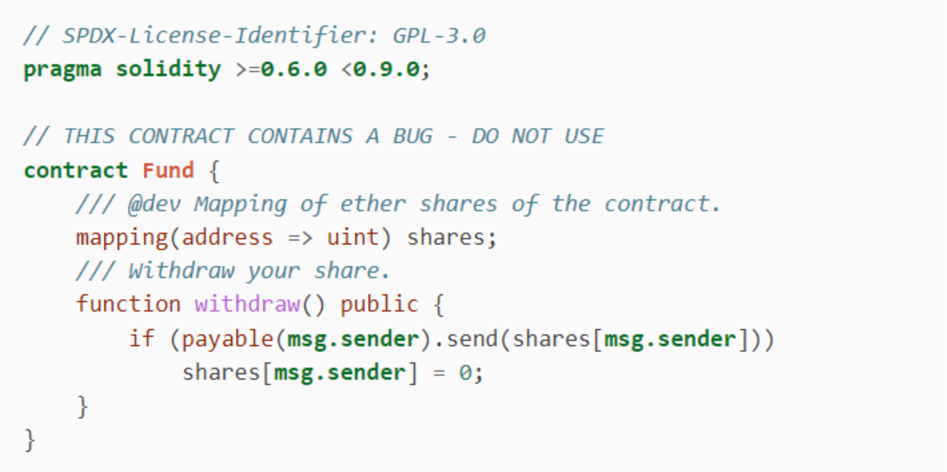
3.1.1 Common exploits

Starting with the common exploits, we found the following to be the most significant, which are explained in more detail over the following sections:

- Re-entracy
- Integer overflow/underflow
- Delegate call
- Denial of service due to unexpected revert
- Transaction ordering dependence
- Timestamp dependence

3.1.1.1 Re-entrancy

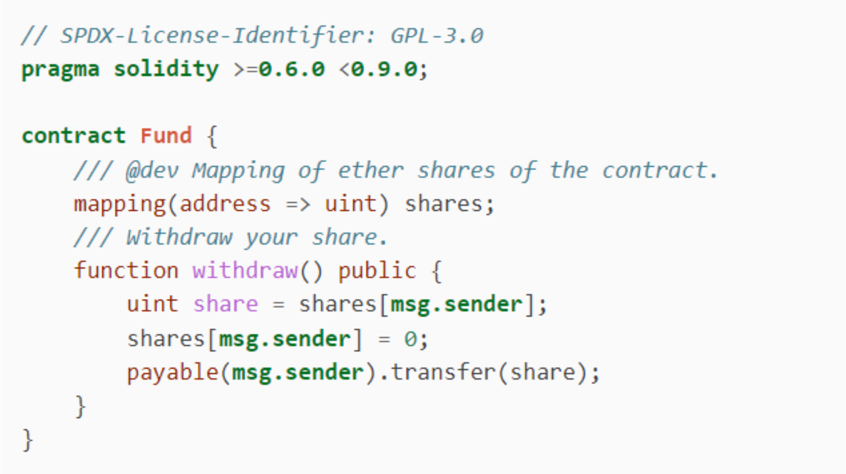
Re-entrancy happens when a contract A calls a contract B which, in turn, calls contract A recursively. The exploit is possible due to an incorrect order of statements. If the external call exists prior to the state change which would prevent it from happening again, it is possible to keep invoking the original method. This is a well-known bug which was the root cause of the famous DAO hack - where 3.6 million Ether were stolen [28]. On figure 11, we can observe a re-entrancy opportunity. The send method could be interacting with a contract which calls back the "withdraw" method until all funds are taken.



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

Fig. 11: Re-entrancy vulnerability example [29]

Figure 12, in turn, mitigates this vulnerability by making use of the Checks-Effects-Interactions pattern. This pattern enforces a certain order in the code. First, checks are executed, then state changes, and, finally, external calls.



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Fund {
    /// @dev Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

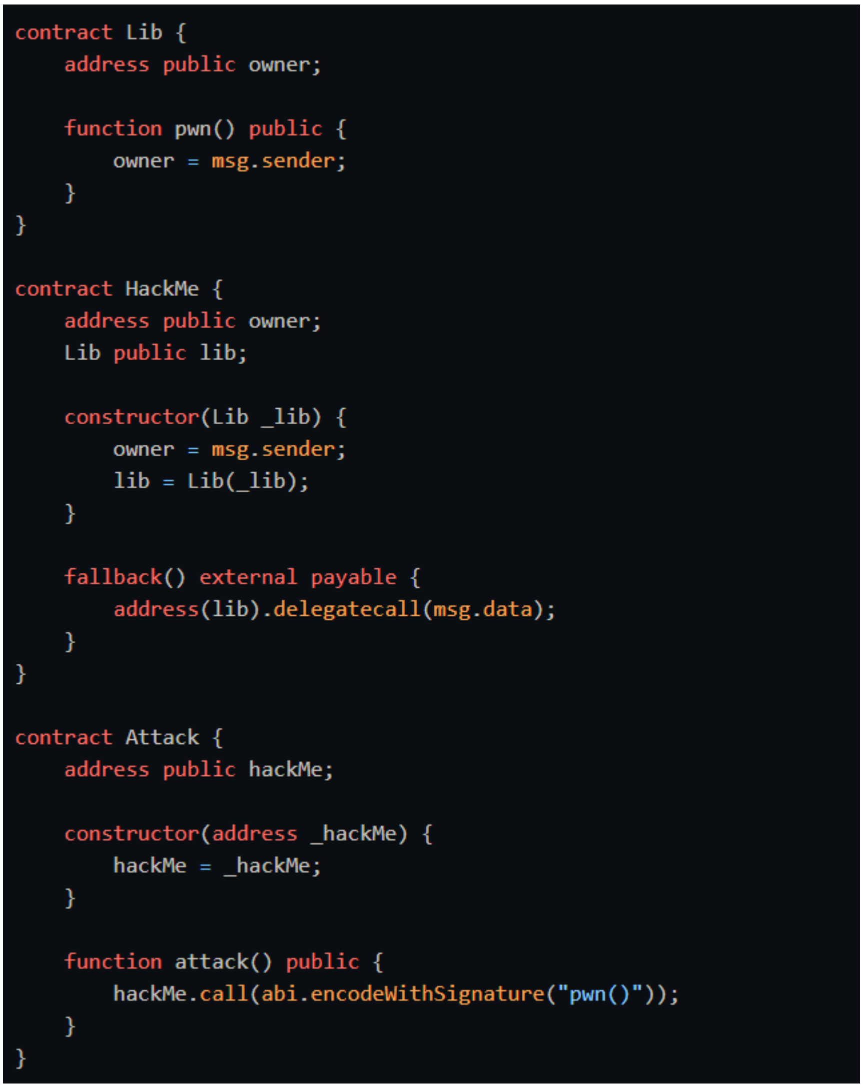
Fig. 12: Mitigating re-entrancy vulnerability [29]

3.1.1.2 Integer overflow/underflow

Before Solidity v0.8.0 (currently at v0.8.17), integer overflow/underflow was a possibility that could go unnoticed and be the cause of a hack. Fortunately, a arithmetic check was put in place on version 0.8.0 which reverts if an integer overflow/underflow occurs [30]. Having that said, this hitch, in general, should no longer be a concern, as the protocols are continuously updated to use the latest version of underlying software.

3.1.1.3 Delegate call

The delegate call method should be carefully used since the developer has to consider that it is a context preserving interaction. In other words, it means, besides preserving the implicit message object, any state changes occur on the caller contract's storage. Figure 13 presents a smart contract which is vulnerable to the delegate call attack. If the contract "Attack" executes the "attack" method, this will trigger the fallback function of the contract "HackMe", ending up calling the method "pwn" of the Lib contract. How was the vulnerability exploited? Well, we just made the "Attack" contract the new owner of the "HackMe" contract. Because the delegatecall operation preserves the context of the caller contract, when the "Lib" contract updates the owner, it is actually updating the owner of the "HackMe" contract. The msg.sender is always the address of the "Attack" contract since the context is preserved. [31]

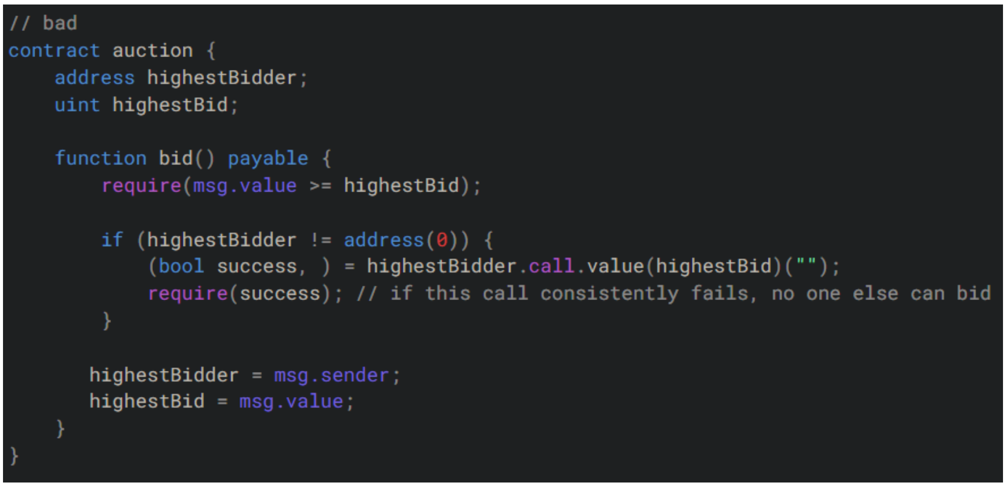


```
contract Lib {
    address public owner;

    function pwn() public {
        owner = msg.sender;
    }
}

contract HackMe {
    address public owner;
    Lib public lib;

    constructor(Lib _lib) {
        owner = msg.sender;
        lib = Lib(_lib);
    }

    fallback() external payable {
        address(lib).delegatecall(msg.data);
    }
}

contract Attack {
    address public hackMe;

    constructor(address _hackMe) {
        hackMe = _hackMe;
    }

    function attack() public {
        hackMe.call(abi.encodeWithSignature("pwn()"));
    }
}
```

Fig. 13: Delegate call vulnerability example [31]

3.1.1.4 Denial of service due to unexpected revert

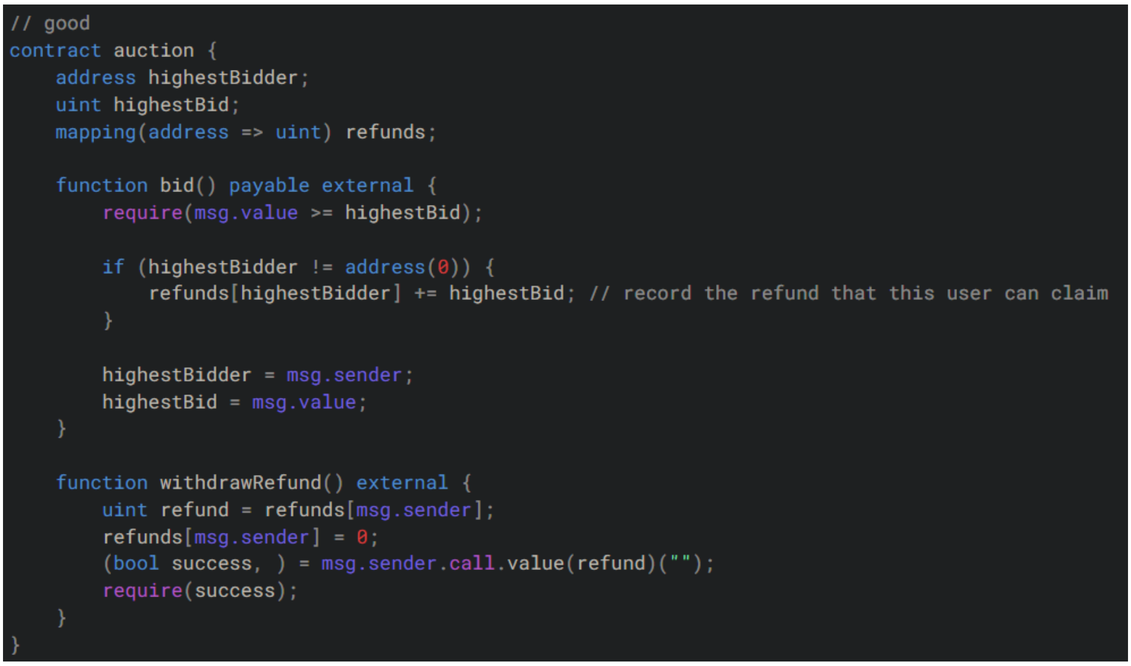
A denial of service can occur when an external call fails hence reverting the transaction. An example of this vulnerability is provided on Figure 14. If the first bidder is a smart contract which reverts upon receiving any payment, then the attacker deliberately denies a higher bidder to take his place.



```
// bad
contract auction {
    address highestBidder;
    uint highestBid;

    function bid() payable {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            (bool success, ) = highestBidder.call.value(highestBid)("");
            require(success); // if this call consistently fails, no one else can bid
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Fig. 14: Denial of service due to unexpected revert example [32]

To mitigate this issue a best practice is to favor pull over push external calls. In other words, allow users to be refunded on demand, instead of mixing push external calls with other business logic. A refactored example solving the above mentioned issue is represented on Figure 15.



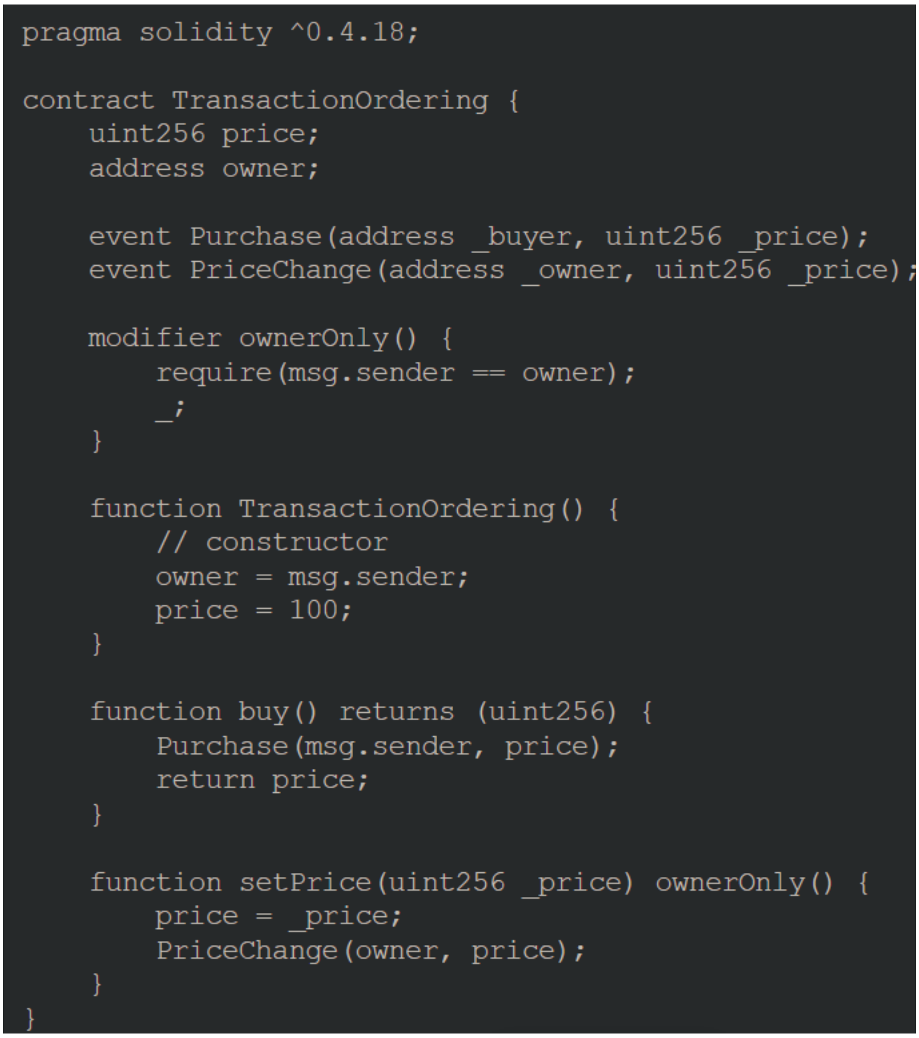
```
// good
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            refunds[highestBidder] += highestBid; // record the refund that this user can claim
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        (bool success, ) = msg.sender.call.value(refund)("");
        require(success);
    }
}
```

Fig. 15: Prevent denial of service due to unexpected revert [32]

3.1.1.5 Transaction ordering dependence

The order on which the transactions are picked up from the mempool often depend on the gas sent with them. Typically, more gas means the transaction will be prioritized. However, miners could potentially reorder transactions for their benefit [27]. For this reason, developers are advised to code in an order agnostic way. To illustrate this vulnerability, we included a mock contract on Figure 16. For simplicity, let's assume that two transactions are sent to the mempool at around the same time.

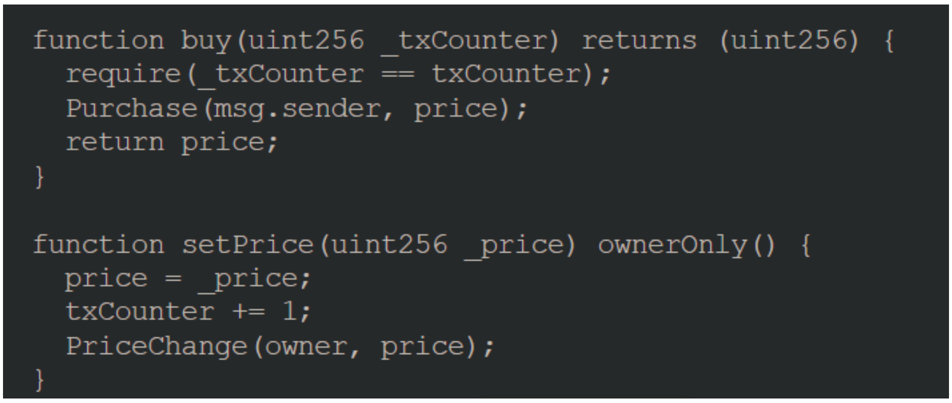
One transaction is an arbitrary user calling the "buy" method, with the other representing the contract's owner altering the price, by using the "setPrice" method. It is evident the order on which these transactions are processed will produce different results, and could possibly be exploited by a malicious miner or even by the contract's owner.



```
pragma solidity ^0.4.18;

contract TransactionOrdering {
    uint256 price;
    address owner;

    event Purchase(address _buyer, uint256 _price);
    event PriceChange(address _owner, uint256 _price);

    modifier ownerOnly() {
        require(msg.sender == owner);
        _;
    }

    function TransactionOrdering() {
        // constructor
        owner = msg.sender;
        price = 100;
    }

    function buy() returns (uint256) {
        Purchase(msg.sender, price);
        return price;
    }

    function setPrice(uint256 _price) ownerOnly() {
        price = _price;
        PriceChange(owner, price);
    }
}
```

Fig. 16: Example of a contract dependent on transaction order [33]

One way to avoid this problem is to introduce a counter which ensures the price getting paid was the one agreed to - Figure 17.

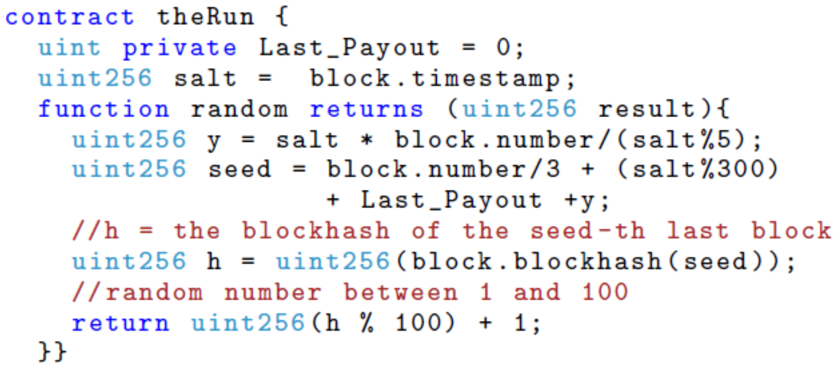


```
function buy(uint256 _txCounter) returns (uint256) {
  require(_txCounter == txCounter);
  Purchase(msg.sender, price);
  return price;
}

function setPrice(uint256 _price) ownerOnly() {
  price = _price;
  txCounter += 1;
  PriceChange(owner, price);
}
```

Fig. 17: Fixing the transaction order vulnerability [33]

3.1.1.6 Timestamp dependence

Block timestamps, like the order of the transactions, can be tweaked by block miners. Developers have been using timestamp as a source of randomness, which is insecure [27]. An example is provided on Figure 18. A miner has the capability of playing back and forth with the final timestamp hence increasing the chances of achieving a random number deserving of a reward.



```
contract theRun {
  uint private Last_Payout = 0;
  uint256 salt =  block.timestamp;
  function random returns (uint256 result){
    uint256 y = salt * block.number/(salt%5);
    uint256 seed = block.number/3 + (salt%300)
                   + Last_Payout +y;
    //h = the blockhash of the seed-th last block
    uint256 h = uint256(block.blockhash(seed));
    //random number between 1 and 100
    return uint256(h % 100) + 1;
  }}
```

Fig. 18: Vulnerable contract which uses block timestamp as source of randomness [27]

3.1.2 Code analysis tools

In the aftermath of many costly hacks, code analysis tools have been surfacing the community. These typically attempt to report bugs on source code [27], while others even provide fixes to the input [34]. Notwithstanding, these emerging tools often fail at identifying all vulnerabilities, providing a dangerous false sense of security to developers [28].

3.2 Optimization

The deployment and execution of smart contracts costs gas, a service fee to cover the resources expended. The more resources used, the more expensive the fee will be. In addition, gas price scales with network congestion. Considering all of this, it is in the best interest of developers to foster design patterns which opt for gas saving techniques. Embracing this mindset will reduce deployment costs as well as transaction costs for the end users, therefore nurturing wider adoption. We have studied and summarised optimization techniques brought up by the R&D community [35].

- Proxy delegate - a pattern to reduce deployment costs and ensure peace of mind regarding protocol upgrades. It relies on the "delegate_call" Solidity method, which executes remote calls using the context of the originator.
- Contract for data - because moving data over the blockchain can be truly costly, an approach is to have a dedicated contract for datawarehousing. As a plus, the business logic of protocols can be changed while the data stays untouched.
- Limit storage - the memory storage option should be used whenever possible.
- Variable packing - take advantage of the compiler's automatic variable packing by declaring variables consecutively. A slot in the EVM can take up to 256 bits, so you could pack two variables of 128 bits, for example. Memory and calldata variables cannot be packed. Further,this does not apply to mappings, structs, and arrays since these always allocate a new slot. [36]
- Marking functions as external - if a function is solely used from outside, make sure to mark it as external to save some gas. This is because external functions extract parameters from the calldata storage structure.
- Prefer internal functions - internal functions' parameters are passed as references, whereas public ones are passed in memory thus more expensive.
- Short constant strings - logically, shorter strings will consume less room hence requiring less gas.
- Freeing storage - freeing storage gives some gas back. Use the delete keyword for this purpose.

3.3 Blockchain scalability

The blockchain trilemma stresses the difficulty of achieving security, decentralization, and scalability on a system. The more decentralized a system is, the more difficult it is to scale it. On the other hand, a highly scalable system tends to sacrifice decentralization or security. For public and decentralized blockchains (e.g. Ethereum) to be ready for mass adoption, the scalability pillar needs to be improved. The Ethereum blockchain can currently only process 15 transactions per second, which has already resulted in network congestion and massive fees. [37]

3.3.1 Sharding

Sharding allows large datasets to be split in chunks and distributed across multiple locations. This concept is aliased as horizontal scaling, due to the addition of new nodes. In turn, vertical scaling concerns increasing the hardware capacitity of a machine (e.g. increase RAM). As in every architectural decision, sharding has benefits and downsides. On the bright side, a sharded system is capable of handling more read/write requests, has larger storage capacity, and offers higher availability. The drawbacks consist of query overhead, maintenance complexity, and bigger infrastructure costs. [38]

For Ethereum, the goal is to have shards interoperating with layer 2 rollups - more on that right away - to reduce network congestion and increase transaction throughput. Furthermore, Ethereum sharding will appeal to more decentralization since hardware requirements to run a node will drastically decrease. With rollups and data availability from sharding, Ethereum envisions a bold number of 100000 (one hundred thousand) transactions per second! [39]

3.3.2 Layer 2 solutions

Layer 2 represents a network or technology which is built on top of blockchain networks with the aim of increasing the scalability of the base layer. These solutions consume and process pending transactions from the main network, hence reducing its transaction overhead. As an example, Bitcoin is a Layer 1 (base layer) network, while the Lightning Network [4] is a Layer 2 solution. [40]

3.3.2.1 Channels

A channel permits groups of users, through unanimous consent, to transact multiple times off-chain. It requires all parties to lock collateral funds on the blockchain. The collateral is deposited on a multisignature smart contract which later will payout all parties involved. [41]

Transactions are signed by every user, and the final one, closes the channel and sends the result to the underlying blockchain, unlocking the collaterized funds. This solution ensures privacy and instant finality [42]. Through this less decentralized approach, high and cheap transaction throughput is achieved. On figure 19, we provide a high level overview of the channel solution.

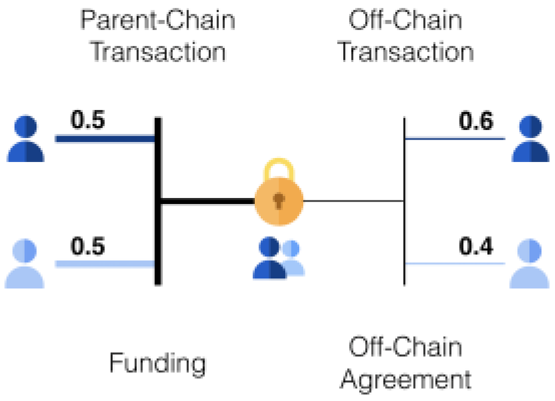


Fig. 19: Channel high level overview [42]

When the number of participants is known, many transactions are required, and all participants are constantly available, channels are a great solution. They can also be used for high-frequency micropayments. On the other hand, they are less ideal for seldom transactions [43]. Figure 20 illustrates one basic use case for this solution. The use case is the "4 in a row" game where there will be a lot of quick moves from the players until the game is finished. By using channels, only the final state - the result of the game - will be propagated to the root chain, hence reducing costs and improving speed.

3.3.2.2 Plasma

Plasma chains, also referred to as commit-chains, are chains anchored to a root chain. This anchoring is achieved using a smart contract on the root chain. These chains are controlled by operators,

[4] http://lightning.network/docs/

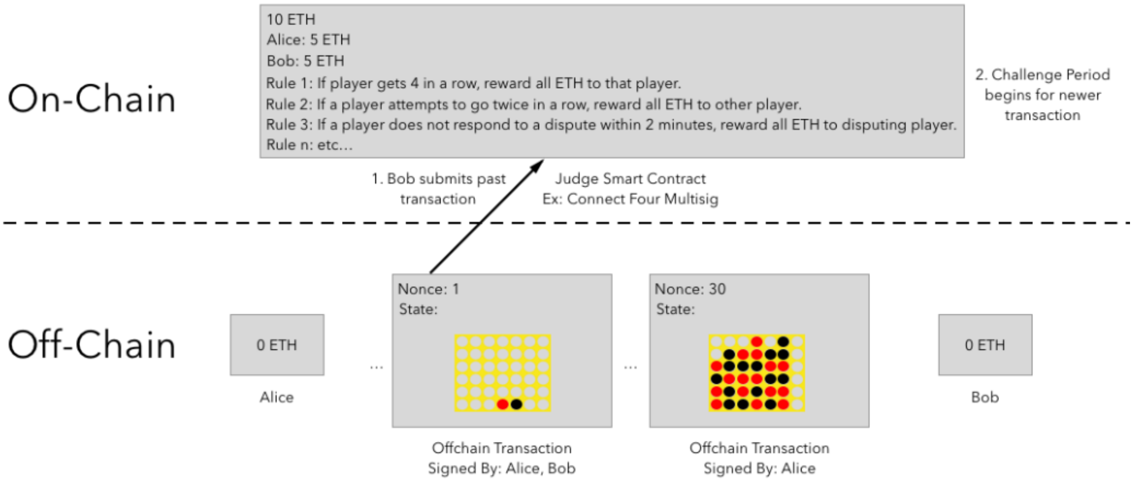

On-Chain
10 ETH
Alice: 5 ETH
Bob: 5 ETH
Rule 1: If player gets 4 in a row, reward all ETH to that player.
Rule 2: If a player attempts to go twice in a row, reward all ETH to other player.
Rule 3: If a player does not respond to a dispute within 2 minutes, reward all ETH to disputing player.
Rule n: etc...
2. Challenge Period begins for newer transaction
1. Bob submits past transaction
Judge Smart Contract
Ex: Connect Four Multisig
Off-Chain
0 ETH
Alice
Nonce: 1
State:
Offchain Transaction
Signed By: Alice, Bob
Nonce: 30
State:
Offchain Transaction
Signed By: Alice
0 ETH
Bob

Fig. 20: Channel basic use case [43]

typically a one person job, who validate transactions and produce blocks [44] . Periodically, the operator commits the state of the chain by sending the blocks' headers to the root chain [45]. This is the reason plasma chains are also known as commit-chains, due to the periodic publishing on the root chain, which assures eventual finality. It is possible to attach multiple plasma chains to a root chain, hence accomplishing a much bigger transaction throughput [45]. Figure 21 represents multiple plasma chains over a root chain [44].

Plasma chains do not have a consensus mechanism as they are controlled by a small amount of operators, typically one. As the finality is achieved on the root chain, plasma chains benefit from the security and decentralization of the base layer. Notwithstanding, fraudulent behavior could be attempted. A challenge period - usually one week - has been introduced to identify frauds, punish attackers, and reward challengers [44].

To enter the chain, funds have to be deposited on the Plasma smart contract residing on the root chain. The operator is then responsible to create the deposited funds on the user's address on the plasma chain. Once the user confirms the funds have been generated, transactions on the plasma chain can be executed [44].

To withdraw funds from the plasma chain, a user sends a withdrawal request to the root chain's Plasma smart contract. Further, depending on the Plasma implementation, the user needs to provide different Merkle proofs. [44]

With plasma chains, high throughput is achieved while relying on the security and decentralization of the root chain. One major drawback is the lack of support for general computations, supporting only basic transactions such as token transfers and swaps [43]. Other downsides regard the operator dependency to store and serve data on demand, and withdrawal delay due to challenge periods.

3.3.2.3 Sidechains

Sidechains are networks, attached to a root chain, which have their own consensus mechanism and validators. With this setup, this solution does not rely on the root chain for security and decentralization. Differently to plasmas, sidechains have no commit checkpoints, which gives no recovery options for users if, for some reason, the sidechain breaks.

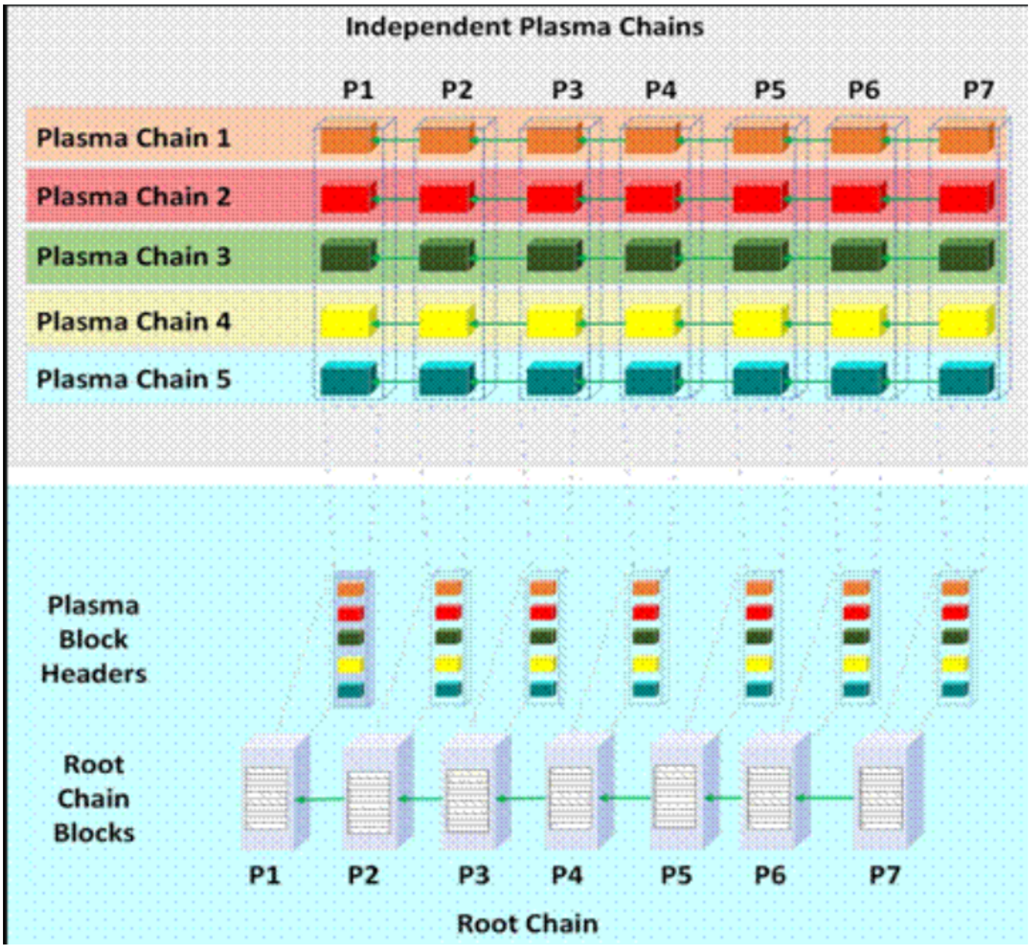


Fig. 21: Mutiple plasma chains over a root chain [45]

Sidechains can use the root chain's assets directly, through an innovative mechanism called two-way peg [42]. The mechanism is based on locks which are unlocked through a Simplified Payment Verification (SPV) proof. There are two varieties of the two-way peg mechanism: symmetric and asymmetric. With the first option, an SPV proof is required to transfer funds between the distinct chains, no matter what direction. The latter, in turn, can be used when users are mindful about the state of the parent chain, enforcing that an SPV proof is solely needed to move funds from the sidechain to the root chain [46].

Sidechains provide higher throughput comparatively to the root chain. Moreover, they allow assets to be moved between chains at the same exchange rate. However, building a sidechain generates more complexity and introduces new attack vectors [42].

Polygon [5] is one of the most promising and developed Ethereum's sidechain solutions. They have implemented a three layer architecture which achieves high throughput while piggybacking to Ethereum's security and decentralization. The Ethereum layer is comprised of a set of contracts on the Ethereum mainnet. Next, the Heimdall layer, where proof-of-stake nodes monitor the staking contracts and commit the status of the Polygon network to checkpoint contracts inhabiting the Ethereum blockchain. Lastly, the Bor layer, represented by a network of nodes running software to publish new blocks [47]. The just described architecture can be seen on Figure 22.

[5] https://polygon.technology/

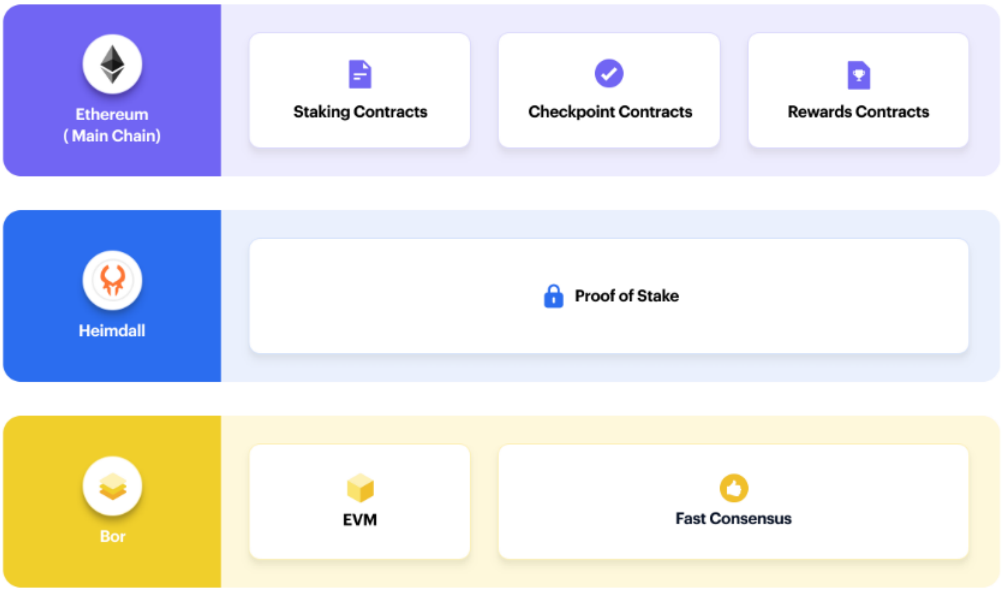


Fig. 22: Polygon's three layer architecture [47]

3.3.2.4 Rollups

Rollups are designed to execute transactions outside of layer 1 blockchains later posting the batched and compressed state of them. Two types of rollups have been developed so far: optimistic and zero-knowledge [48].

Optimistic rollups are similar to Plasmas in many ways. They have operators running the network and they have challenge periods to catch any misbehavior. Distinctly, they post, even though compressed, all transaction data on the root chain, increasing the transparency. Besides that, optimistic rollups are compatible with EVM and Solidity, thus allowing developers to seamlessly port over smart contracts designed for Ethereum [49].

In the same way as optimistic rollups, zero-knowledge rollups are controlled by operators. Operators are still responsible for processing the transactions and batching them into the root chain. However, with zero-knowledge rollups, only validity proofs are posted, instead of all data, which is the case for optimistic rollups. Validity proofs allow verification of statements without revealing them, hence inferring the name zero-knowledge proofs. Zero-knowledge rollups employ validity proofs which guarantee the correct execution of state transitions without having to re-execute transactions on the root chain. On the downside, zero-knowledge rollups do not support general computation, and are more resource intensive than optimistic rollups, due to the crafting of validity proofs [50].

3.3.2.5 Overview

To summarize layer 2 solutions, we have provided table 1 where we compare openness, finality, security, and support for general computation.

3.4 Royalties on blockchain protocols

A great use case for NFTs is to ensure continuous and automatic royalty payment to creators [23]. This could theoretically be achieved by defining the royalty configuration upon NFT creation, and

Table 1: Overview of layer 2 solutions

	Channels	Plasmas	Sidechains	Optimistic rollups	ZK rollups
Open to arbitrary users	No	Yes	Yes	Yes	Yes
Security comes from	Root chain	Root chain	Sidechain	Root chain	Root chain
Finality	Instant	Eventual (once challenge period ends)	Instant	Eventual (once challenge period ends)	Instant
Supports general computation	Yes	No	Yes	Yes	No

then relying on the on-chain token transfer mechanism for the payments. However, this is currently not provided by default on token transfers. Even though it would be useful, it is not possible to understand if a token transfer is the result of a sale, or simply of a user moving assets between wallets.

In hope of standardizing and nurturing royalty payment adoption, the Ethereum Improvement Proposal 2981 has emerged. This standard provides an interface to define and share royalty information of an NFT [51]. With this information at hand, it is up to the marketplaces to implement the actual payment, which does not necessarily mean it has to be an on-chain payment. The creator should proactively educate himself about how royalties are processed on the marketplace in question.

3.5 Summary

In this chapter, we have dived into optimization and security techniques which apply when writing smart contracts in Solidity. We have covered the common exploits of the language and of the underlying consensus mechanism. Afterwards, we had a look at ways to scale blockchains, introducing sharding, and layer 2 solutions: Channels; Plasma; Sidechains; Rollups. Finally, it was described how NFTs' royalties can be handled on blockchain protocols.

The next chapter references the work of other authors which had the same intent to mitigate ticket scalping.

4 Related work

Over the years, different solutions have been proposed to mitigate ticket scalping. Throughout the following lines, we describe a couple of them.

One of the proposals introduces a centralized exchange (CE) that randomly assigns tickets in the primary market [3]. In addition, ticket owners can be partially refunded when returning a ticket. The returned ticket is then randomly assigned to a waiting-queue. Moreover, the ticket ownership is tracked through ledger records and user identification is necessary upon usage. A ticket that is bought outside the exchange in question is worthless to anyone except its last owner as reported on the ledger. The randomness embedded in this solution highly decreases the chance of scalpers acquiring tickets. Furthermore, the asset owner tracking feature completely disables the possibility to trade these assets in secondary markets. However, a scalper could still use bots to occupy multiple positions in the primary market and/or on the waiting queue, increasing his chances to randomly acquiring the ticket, even though there would be a minor profit. In general, our outlook is that this is a very robust and complete proposal.

A different proposal suggests a system with batch assignments given a couple of allocation periods [2]. A few slots are opened for applications during a certain amount of time, let's say 1 day. Once that period ends, the slots are allocated for the applicants. If there are more applications than slots then they are randomly assigned. Cancelled slots are postponed to the next batch. The use of allocation periods removes the speed advantage that bots give to scalpers on a traditional first-come-first-served system.

4.1 Solutions using Ethereum

TickEth is a system that uses Ethereum in order to mitigate ticketing industry problems, more specifically the inability of checking the authenticity of tickets sold online and the secondary market unwieldy price difference [52].

The supra-mentioned system features ticket purchase, secondary market, refund, address change and the possibility to allow authorized entities for reselling. Contrarily to our system, they do not make use of NFTs for changing ticket ownership, and instead issue a new ticket and deactivate the old one on a ticket resell. Moreover, they include a database server to keep track of the tickets that have already been used for an event. Albeit quite promising and having interesting details, the system in question was a prototype and was not properly tested nor audited. In addition, a few characteristics were not very transparent, such as the QR code functionality. What does the QR code represent? How does the end-user have access to it? Finally, the event object deployed in the smart contract did not have a property for the number of tickets, so in theory there was no upper boundary on how many tickets could be sold for an event.

A research-driven approach implemented a proof-of-concept NFT use case for event ticketing application. Using Ethereum's smart contracts, they developed a fully decentralized system that allows for peer-to-peer ticket transactions [53].

Figure 23 represents the UML Diagram of the system. In short, for each new event a new smart contract is deployed on the blockchain network. This contract implements the ERC-721 standard, and therefore allows for the use of NFTs. There are only two agents in the system, the event organizer and the event attendees, and to do business they just need to interact with the smart-contract - no intermediary is necessary.

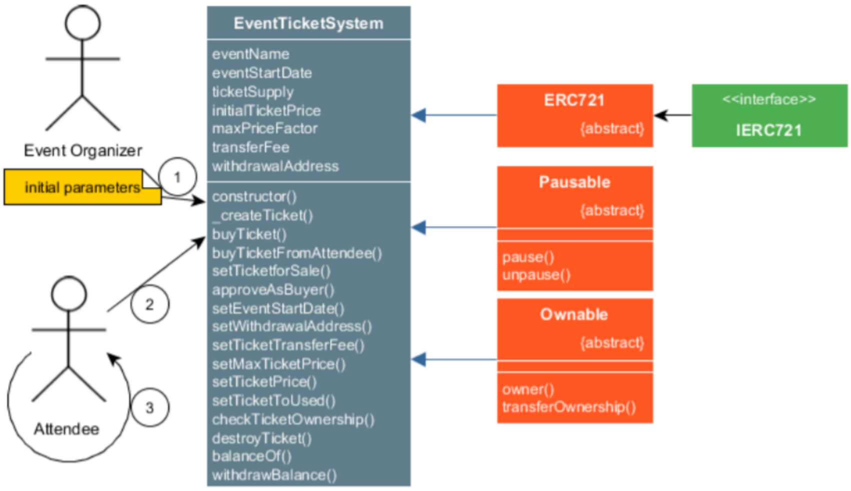


Fig. 23: System's UML Diagram [53]

The event organizer creates the new event, and can later change event parameters, pause the selling of tickets and withdraw contract's balance. On the other hand, attendees can buy tickets or set them to be sold, with the maximum price being the one defined by the event owner.

Anyone who owns an Ethereum wallet can use the system, either by interacting directly with the smart-contract, or by making use of interfaces such as OpenSea, which feature an NFT Marketplace.

The authors achieved digitization, secondary markets, independence, security, validation, transparency, automation and cost efficiency. However, at the time of writing, cost efficiency of the system would need to be reviewed, because Ethereum's gas fees have drastically increased, and we believe deploying a smart contract for each new event would end up being unsustainable. To complement, the approach to set the ticket to used - invokes a smart contract method to change data - could introduce delayed venue verification since writing to the blockchain is slow.

5 Development

Upon starting developing the solution, we realized the development approach previously proposed would have to be rethought. Instead of developing the interface, then the contracts and finally implement the connection between the prior two, we opted for a requirement-driven approach. In other words, the idea was to work on a requirement from beginning to end before moving to the next one. With this change, we would achieve faster iterations, releasing in modules, and for that reason facilitate code review tasks. Moreover, due to the dimension and complexity of the project, the effort pertained solely on the implementation and optimization of the protocol's smart contracts. Back-end and front-end changes were assigned to other members of the team to reflect the changes delivered by us at the protocol level.

5.1 Requirements

Before actually starting with the development, we had to thoroughly understand the system's requirements. For that matter, we listed them below.

- The system should allow, at any time, event producers to control marketplace trading availability
    - The system should allow event producers to enable marketplace trading
    - The system should allow event producers to disable marketplace trading
- The system should allow event producers to define a marketplace trading margin, in percentage
- The system should block ticket reselling while the event is on-going, for any of the following conditions:
    - Marketplace trading is disabled
    - Trade is attempted from a foreign marketplace
    - Selling price exceeds the margin boundary
- The system should allow free trading, after the event happened
- The system should allow event producers to define and modify the trading royalty, which includes:
    - Percentage variation
    - Fixed token amount
    - Receiver address
- The system should allow event producers to remove a predefined trading royalty
- The system should pay the royalty to the receiver, upon any token transfer, if the royalty was defined
    - The system should pay the biggest value between the percentage variation and fixed token amount
- The system should allow users to set their tickets to sale
    - The system should allow users to define a selling price

- The system should allow users to remove their tickets from sale
- The system should allow users to buy tickets which are on sale

5.2 Marketplace implementation

Originally, the plan was to hand-craft our marketplace - which had some development already in place - to deal with the NFT tickets and its particularities. However, the company this project was being developed with was time constrained with employing a marketplace that could handle generic NFTs. With that in mind, we had a few meetings to discuss possibilities, mainly how to use a generic NFT marketplace which would comply with the ticket specific properties. It was key that on-chain price validation was still put in place to prevent ticket scalping. In the end, the Marketplace was delivered as a separate set of contracts and allowed fixed price trading as well as auctions. It was based on Tatum's open-source contracts [6]. The event specific trading constraints were attached to the Event contract, therefore providing loose coupling between the Marketplace and Event implementations.

It is earnest to refer that only a minor contribution to the Marketplace's contracts was given by the author of this thesis. Instead, the effort was reallocated to work on the Event implementation, where the trading restrictions would really take place. At any rate, and very briefly, the Marketplace contracts allow an arbitrary user to set NFTs to sale, remove them from sale, or buy an NFT which is on sale.

5.3 Ticket trading rules

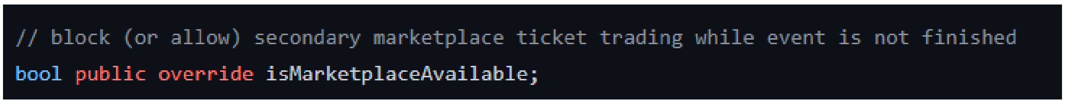
The requirements state two ticket trading rules which apply before the event takes place:

- Trading availability
- Price margin

Once the event has ended, no trading restrictions will be enforced, to allow for *a posteriori* price speculation.

5.3.1 Trading availability

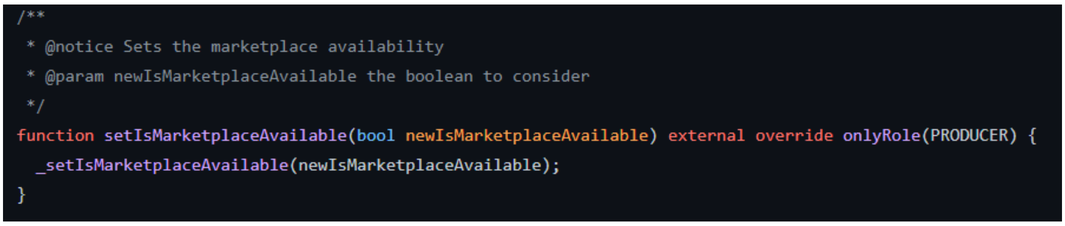
The event producer has the possibility to switch the trading availability for a given event. This was a simple addition to the protocol, where we added a publicly accessible variable (Figure 24) and a method to alter it (Figure 25).



```
// block (or allow) secondary marketplace ticket trading while event is not finished
bool public override isMarketplaceAvailable;
```

Fig. 24: Definition of marketplace availability variable

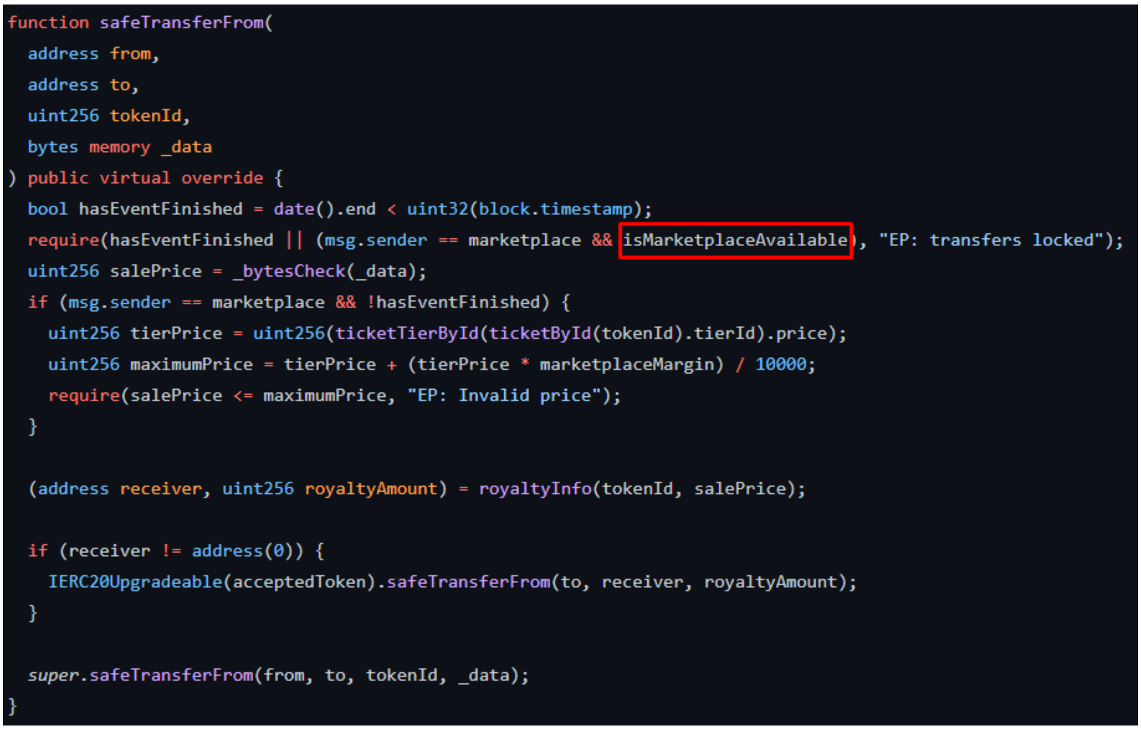
[6] https://github.com/tatumio/smart-contracts/tree/master/contracts/tatum/nft



```
/**
 * @notice Sets the marketplace availability
 * @param newIsMarketplaceAvailable the boolean to consider
 */
function setIsMarketplaceAvailable(bool newIsMarketplaceAvailable) external override onlyRole(PRODUCER) {
  _setIsMarketplaceAvailable(newIsMarketplaceAvailable);
}
```

Fig. 25: Definition of method to modify marketplace availability

Upon a transfer attempt, the protocol will autonomously verify if the marketplace is available and revert the transfer otherwise (Figure 26).



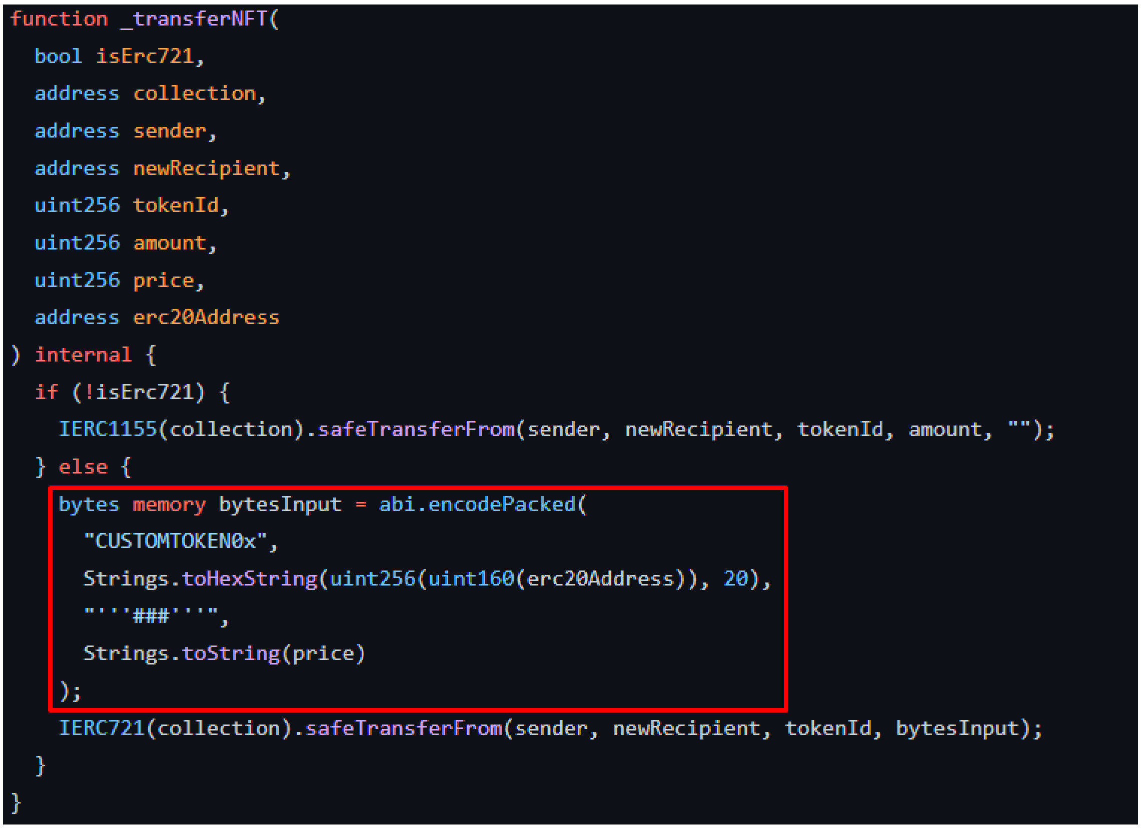
```
function safeTransferFrom(
  address from,
  address to,
  uint256 tokenId,
  bytes memory _data
) public virtual override {
  bool hasEventFinished = date().end < uint32(block.timestamp);
  require(hasEventFinished || (msg.sender == marketplace && isMarketplaceAvailable), "EP: transfers locked");
  uint256 salePrice = _bytesCheck(_data);
  if (msg.sender == marketplace && !hasEventFinished) {
    uint256 tierPrice = uint256(ticketTierById(ticketById(tokenId).tierId).price);
    uint256 maximumPrice = tierPrice + (tierPrice * marketplaceMargin) / 10000;
    require(salePrice <= maximumPrice, "EP: Invalid price");
  }

  (address receiver, uint256 royaltyAmount) = royaltyInfo(tokenId, salePrice);

  if (receiver != address(0)) {
    IERC20Upgradeable(acceptedToken).safeTransferFrom(to, receiver, royaltyAmount);
  }

  super.safeTransferFrom(from, to, tokenId, _data);
}
```

Fig. 26: Verification of marketplace availability upon transfer

5.3.2 Marketplace margin

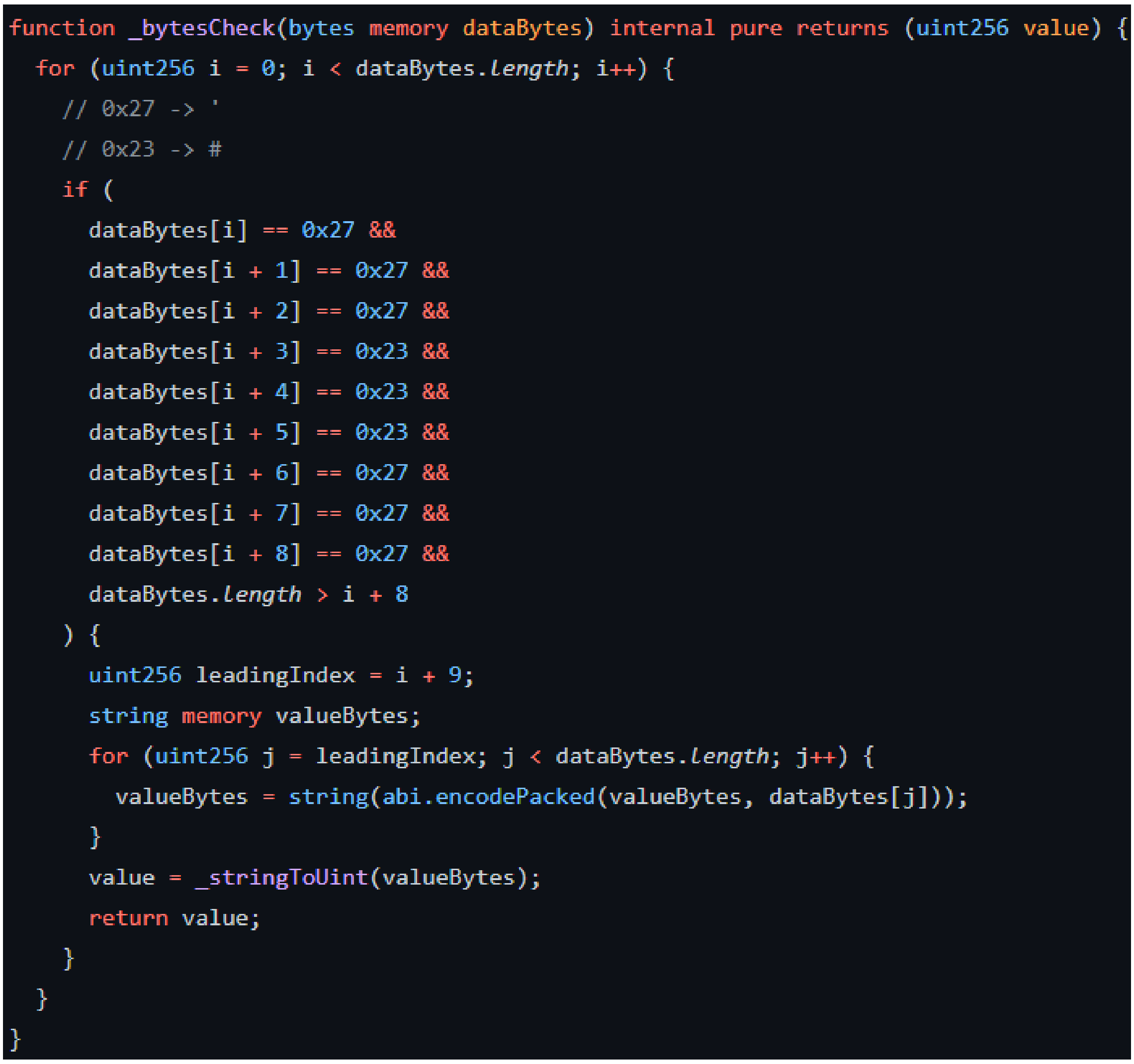
The marketplace margin is a percentage value defined by event producers and it's used as a trading price boundary. The basis price of a ticket depends on its tier. As an example, for a marketplace margin of 10% and a tier price of 1 token, the maximum allowed trading price would be 1,1 tokens. The challenge with this requirement was that we had to have the information on the selling price to be able to validate it. It was a matter of utter importance, because if this was not properly put in place, we risked uncontrolled price manipulation thus failing in our vision to mitigate ticket scalping. Fortunately, one of the safeTransferMethod methods of the ERC721 standard can receive a fourth parameter. With that, we are able to provide extra metadata, such as the selling price.

The protocol's marketplace encodes the selling price in a specific format (Figure 27) which is later decoded on the Event contract in order to validate it (Figure 28).



```
function _transferNFT(
  bool isErc721,
  address collection,
  address sender,
  address newRecipient,
  uint256 tokenId,
  uint256 amount,
  uint256 price,
  address erc20Address
) internal {
  if (!isErc721) {
    IERC1155(collection).safeTransferFrom(sender, newRecipient, tokenId, amount, "");
  } else {
    bytes memory bytesInput = abi.encodePacked(
      "CUSTOMTOKEN0x",
      Strings.toHexString(uint256(uint160(erc20Address)), 20),
      "'''###'''",
      Strings.toString(price)
    );
    IERC721(collection).safeTransferFrom(sender, newRecipient, tokenId, bytesInput);
  }
}
```

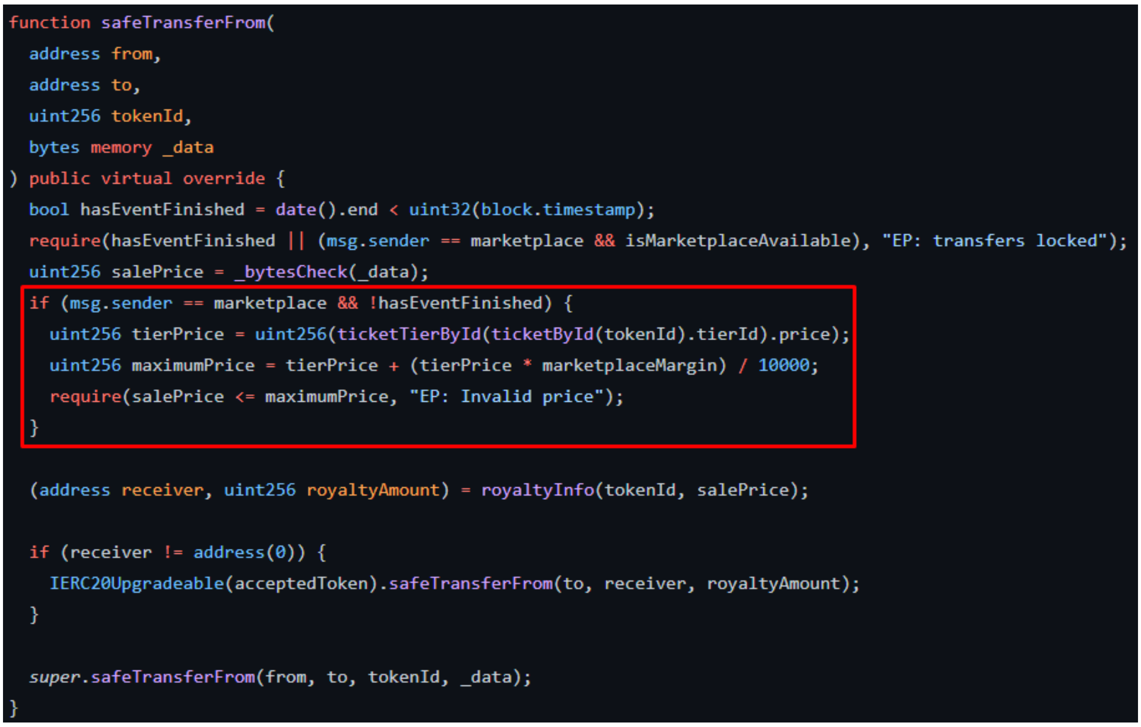
Fig. 27: Marketplace encoding in specific format



```
function _bytesCheck(bytes memory dataBytes) internal pure returns (uint256 value) {
  for (uint256 i = 0; i < dataBytes.length; i++) {
    // 0x27 -> '
    // 0x23 -> #
    if (
      dataBytes[i] == 0x27 &&
      dataBytes[i + 1] == 0x27 &&
      dataBytes[i + 2] == 0x27 &&
      dataBytes[i + 3] == 0x23 &&
      dataBytes[i + 4] == 0x23 &&
      dataBytes[i + 5] == 0x23 &&
      dataBytes[i + 6] == 0x27 &&
      dataBytes[i + 7] == 0x27 &&
      dataBytes[i + 8] == 0x27 &&
      dataBytes.length > i + 8
    ) {
      uint256 leadingIndex = i + 9;
      string memory valueBytes;
      for (uint256 j = leadingIndex; j < dataBytes.length; j++) {
        valueBytes = string(abi.encodePacked(valueBytes, dataBytes[j]));
      }
      value = _stringToUint(valueBytes);
      return value;
    }
  }
}
```

Fig. 28: Event price decoding

The price decoding and validation is performed on the safeTransferFrom method, as presented in figure 29. The check itself is simple, we start by grasping the ticket price based on its tier and then we evaluate if the price of sale is lower or equal to the maximum allowed price.



```
function safeTransferFrom(
  address from,
  address to,
  uint256 tokenId,
  bytes memory _data
) public virtual override {
  bool hasEventFinished = date().end < uint32(block.timestamp);
  require(hasEventFinished || (msg.sender == marketplace && isMarketplaceAvailable), "EP: transfers locked");
  uint256 salePrice = _bytesCheck(_data);
  if (msg.sender == marketplace && !hasEventFinished) {
    uint256 tierPrice = uint256(ticketTierById(ticketById(tokenId).tierId).price);
    uint256 maximumPrice = tierPrice + (tierPrice * marketplaceMargin) / 10000;
    require(salePrice <= maximumPrice, "EP: Invalid price");
  }

  (address receiver, uint256 royaltyAmount) = royaltyInfo(tokenId, salePrice);

  if (receiver != address(0)) {
    IERC20Upgradeable(acceptedToken).safeTransferFrom(to, receiver, royaltyAmount);
  }

  super.safeTransferFrom(from, to, tokenId, _data);
}
```

Fig. 29: Verification of marketplace margin upon transfer

5.3.2.1 The centralization trade-off

There is no guarantee that we will receive the truthfully selling price from third-party marketplaces. They might not know that we expect it, or they could try to be dishonest towards the trade. With that said, to ensure the business requirements were met, we had to centralize our validation. Before the event, we only consider and validate transactions arriving from the marketplace designed by us. All other trading attempts will be ignored. This centralization can feel like a step back since the idea of deploying on a decentralized blockchain is to provide complete decentralization. However, it was the only way, to the best of our knowledge, in which we could extinguish ticket scalping. Further, the protocol will be open sourced for any skeptic player to check out.

5.4 NFTs' royalties

NFTs' royalties were the next requirement to deliver. Royalties are fees to be paid to a given address upon each token transfer. The community has introduced different approaches to handle royalties, which were detailed on Chapter 3.

For this project, according to stakeholders needs and decisions, we went with the on-chain implementation. Authorized users have great flexibility on this topic as they are able to define the royalty receiver, update the royalty variables (percentage and fixed value), and choose a receiver address. Furthermore, they can decide not to collect royalties at all.

Fixed versus percentage royalty price - the receiver will be rewarded with the biggest value between fixed versus percentage price. Let's take the following example:

- Fixed price: 1 (token)

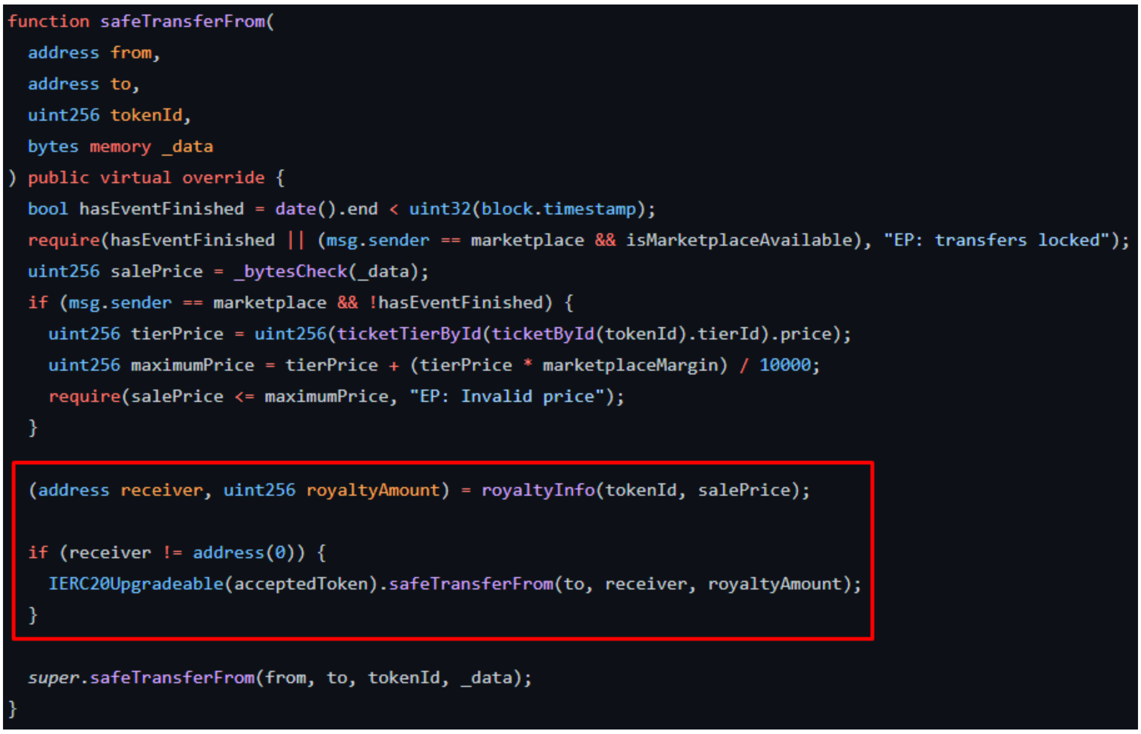
- Percentage: 10

Study case 1: Customer buys ticket for 5 ETH. The percentage computation will return 0.5 ETH while the fixed price remains 1. The fixed price is used.

Study case 2: Customer buys ticket for 11 ETH. The percentage computation will return 1.1 ETH while the fixed price remains 1. Percentage price is used.

With the specifications crystal clear, it was now time to implement the aforementioned. The first approach embedded all the royalty configuration and storage in the Event contract. But we suddenly hit a bottleneck! Our Event contract hit a size bigger than 24kB - the maximum supported by the EVM at the time of writing. In the aftermath, the architectural decision was to move the royalty implementation to a separate contract. Similar to adding a new database to our infrastructure, this new contract permitted the registry and maintenance of royalty's metadata for Event contracts. It was named RoyaltyRegistry.

From the Event contract's point of view, we simply had to query the RoyaltyRegistry for the royalty payment at any token transfer. If no royalty was defined, we skipped the payment.(Figure 30)

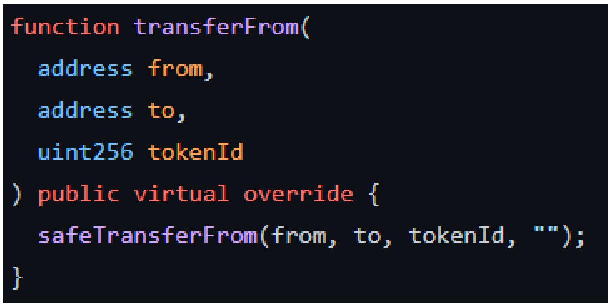


```
function safeTransferFrom(
  address from,
  address to,
  uint256 tokenId,
  bytes memory _data
) public virtual override {
  bool hasEventFinished = date().end < uint32(block.timestamp);
  require(hasEventFinished || (msg.sender == marketplace && isMarketplaceAvailable), "EP: transfers locked");
  uint256 salePrice = _bytesCheck(_data);
  if (msg.sender == marketplace && !hasEventFinished) {
    uint256 tierPrice = uint256(ticketTierById(ticketById(tokenId).tierId).price);
    uint256 maximumPrice = tierPrice + (tierPrice * marketplaceMargin) / 10000;
    require(salePrice <= maximumPrice, "EP: Invalid price");
  }

  (address receiver, uint256 royaltyAmount) = royaltyInfo(tokenId, salePrice);

  if (receiver != address(0)) {
    IERC20Upgradeable(acceptedToken).safeTransferFrom(to, receiver, royaltyAmount);
  }

  super.safeTransferFrom(from, to, tokenId, _data);
}
```

Fig. 30: Royalty payment at token transfer

5.5 Preventing rules bypass

At this point, all the trading validations were in place, and all of them converged to the ERC721 standard's method safeTransferFrom. A perused look at the codebase revealed that a meticulous attacker could easily bypass all these trading validations simply by using another method from the ERC721 standard - transferFrom. To eliminate this threat, an override of the above-mentioned method was included on the Event contract (Figure 31).



```
function transferFrom(
  address from,
  address to,
  uint256 tokenId
) public virtual override {
  safeTransferFrom(from, to, tokenId, "");
}
```

Fig. 31: Override of transferFrom method

5.6 Summary

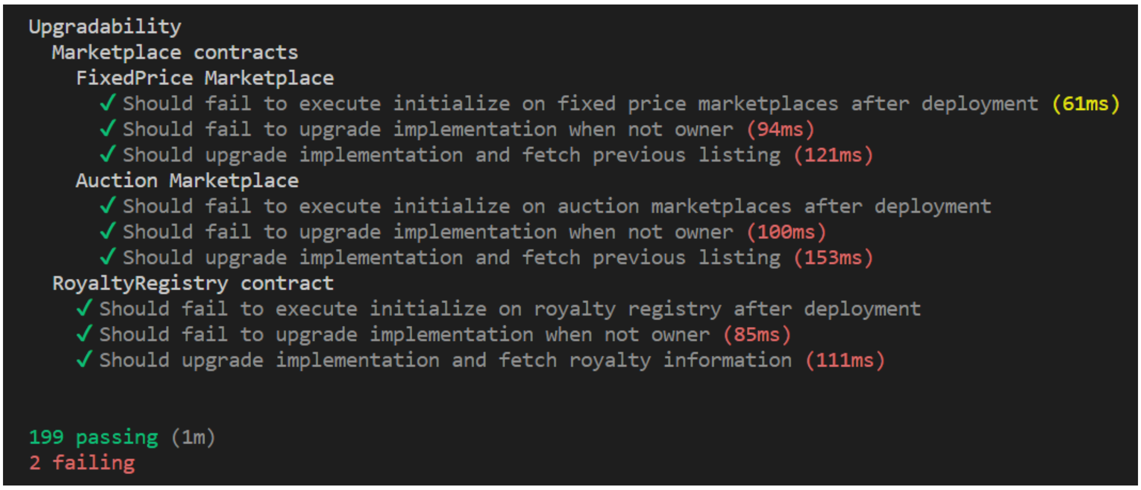
This chapter thoroughly documented the development of the project including pitfalls and subsequent overcomes. It started with a detailed glance over the business requirements, then moving to technical details of the implementation. The majority of the business logic converged to an override applied on one of the methods of the ERC721 standard - safeTransferFrom. With that override, using a centralized design, we ensure ticket reselling complies to the event producer configuration. To complement, there is where we execute the payment of royalties.

6 Testing and Validation

Testing is an important phase of the development life cycle. It can help identify bugs and collect feedback. Ultimately, it should ensure the shipment of higher quality software. For our context, we focused only on unit tests. Unit tests are low level modular tests which are used to test isolated functionality and are cheap to run [54].

6.1 Testing approach

To fulfill testing, we have combined the capabilities of Hardhat with Ethers, Typechain, Mocha, and Chai. When tests are run against this testing environment, an output is generated declaring which tests passed and which ones failed. An example of a report is provided on figure 32.



```
Upgradability
  Marketplace contracts
    FixedPrice Marketplace
      ✓ Should fail to execute initialize on fixed price marketplaces after deployment (61ms)
      ✓ Should fail to upgrade implementation when not owner (94ms)
      ✓ Should upgrade implementation and fetch previous listing (121ms)
    Auction Marketplace
      ✓ Should fail to execute initialize on auction marketplaces after deployment
      ✓ Should fail to upgrade implementation when not owner (100ms)
      ✓ Should upgrade implementation and fetch previous listing (153ms)
  RoyaltyRegistry contract
    ✓ Should fail to execute initialize on royalty registry after deployment
    ✓ Should fail to upgrade implementation when not owner (85ms)
    ✓ Should upgrade implementation and fetch royalty information (111ms)


199 passing (1m)
2 failing
```

Fig. 32: Example of a test report

Having that said, we will now explain in more detail how this environment functions. This will be followed by an exposition of the test cases covered.

6.1.1 Setting up the testing environment

The testing environment is comprised of the aforementioned tools. Each of them serves a different purpose but their synergies make the most of testing. A brief description of these tools is provided below.

- Hardhat - Hardhat[7] is a development environment for Ethereum based software. It provides different components for editing, compiling, debugging and deploying smart contracts.
- Ethers - Ethers[8] is a JavaScript library for connecting and interacting with the Ethereum blockchain.
- Typechain - Typechain[9] will generate the TypeScript types for the contracts based on a blockchain connector. In our case, the connector is Ethers.

[7] https://hardhat.org/hardhat-runner/docs/getting-started#overview

[8] https://docs.ethers.org/v5/

[9] https://github.com/dethcrypto/TypeChain

- Mocha - Mocha[10] is a JavaScript testing framework which supports asynchronous testing.

- Chai - Chai[11] is an assertion library which can be combined with any JavaScript testing framework. In our case, we combined it with Mocha.

Figure 33 depicts the Hardhat environment. It is enhanced with four plugins: Ethers, Typechain, Mocha, and Chai. We have labeled components which can be defined on Hardhat as entities, which are comprised of: contracts, types, tests, and scripts. Finally, we represent the core commands and its outputs: the compile command generates the types, the test command runs the tests and outputs a report, and the deploy command adds the contracts to an Ethereum network of choice (local, testnet, or public).

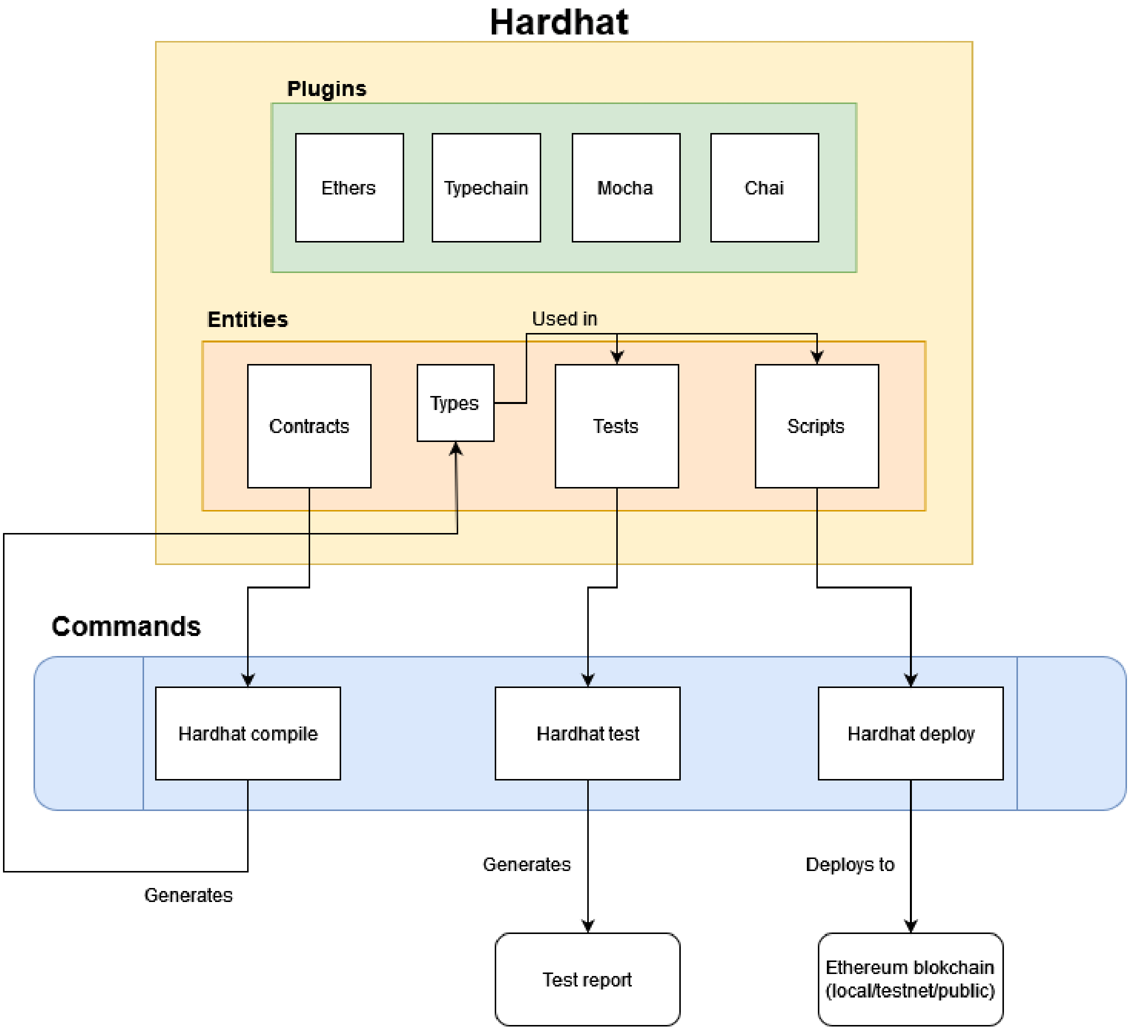


Fig. 33: Hardhat environment

[10] https://mochajs.org/
[11] https://www.chaijs.com/

6.1.2 Basic understanding

On this block, we have documented core concepts/operations for a basic understanding of working with Hardhat and the plugins in question.

External owned accounts

In our environment, we have access to external owned accounts through the Ethers library. Hardhat's ethers plugin extends ethers by providing extra helper functions [55]. For example, we have the function getSigners which gives us random external owned accounts to be used throughout development. Besides that, ethers enables us to instantiate new wallets or retrieve existing ones from private keys, mnemonics, or encrypted json [56].

How to deploy a contract

Making use of Hardhat's ethers plugin [55], deploying a contract is rather easy. We simply execute function deployContract.

How to interact with a contract: read and write

To interact with a contract we would need to have the ABI (Application Binary Interface) and the contract's address. In our context, the ABI and typing is ensured by ethers and typechain, respectively. We can have the contract's address being returned from a deployment, or we might already know it in advance.

Reading data from the blockchain does not cost gas and it is as simple as executing the function of the contract. This becomes trivial if we have typings. An example, extracted from our codebase, is provided on listing 1, where event is the contract and ticketById is the exposed method.

```
const ticket = await event.ticketById(1)
```

Listing 1: Read from blockchain

To write to the blockchain, we would need an account to sign the transaction. Ethers provides a method called connect which establishes the connection of an account to a contract. The remainder of the logic is similar to reading data, where we execute the publicly available methods. Listing 2 exemplifies one operation where a user would change the blokchain's state by interacting with a contract's method.

```
event.connect(yacoobaAdmin).buyTicket([1], secondSignerAddress)
```

Listing 2: Write to the blockchain

How to assert values, exceptions, or emitted events

As explained above, Mocha and Chai have been combined to define our testing framework. Mocha supports asynchronous operations while Chai brings us the assertion modules. An ethereum-specific Chai plugin is provided by Hardhat [57], giving us new pertinent matchers. We provide below three listings: 3,4,5, representing examples on how to assert values, exceptions, and emitted events, respectively. The "revertedWith" and "emit" matchers are some examples that Hardhat's Chai plugin provides.

```
expect(await event.ownerOf(ticketId)).to.equal(thirdSignerAddress)
```

Listing 3: Assert value

```
await expect(event.connect(yacoobaAdmin).buyTicket([1, 1],
    secondSignerAddress)).to.be.revertedWith(
  'ET: sold out',
)
```

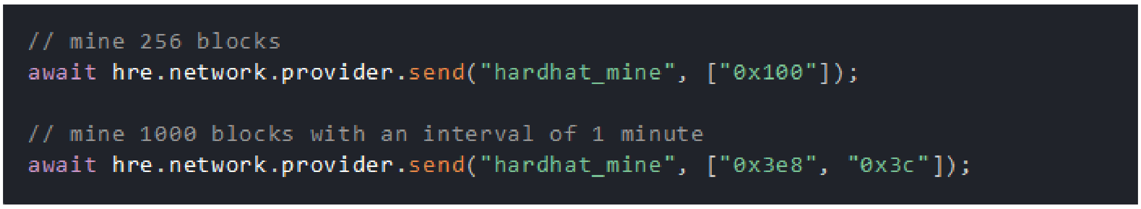
Listing 4: Assert exception

```
await expect(event.connect(yacoobaAdmin).setFee(newFee)).to.emit(event,
    'ProtocolFeeUpdated')
```

Listing 5: Assert event

How to advance time

Hardhat allows us to advance time in the hardhat network, which is quite handy when we want to test time-based operations [58]. An example from their documentation is shown below, on Figure 34.



```
// mine 256 blocks
await hre.network.provider.send("hardhat_mine", ["0x100"]);

// mine 1000 blocks with an interval of 1 minute
await hre.network.provider.send("hardhat_mine", ["0x3e8", "0x3c"]);
```

Fig. 34: Hardhat mine block function [58]

6.2 Test cases covered

In total, there were 169 unit tests covering the protocol. Grouped by context, the EventManager contract owns the biggest chunk of them, with 39 unit tests. Then, by decreasing order, we have Auction, EventTickets, Forwarder, FixedPrice, EventsFactory, and Event, with 32, 27, 25, 22, 18, and 6 unit tests, respectively. A visual representation is displayed on Table 2.

Context	Count
EventManager	39
Auction	32
EventTickets	27
Forwarder	25
FixedPrice	22
EventsFactory	18
Event	6
Total	169

Table 2: Count of unit tests by context

The full list of unit tests can be found in Appendix A. We will move on to explain the tests created by us which verify our functionality is working neatly. These are directed to the EventTickets contract.

6.2.1 Ticket transfer

Making sure ticket transfer is complying to the business requirements is of utmost importance to mitigate ticket scalping. We have organized the corresponding tests according to the type of constraint, namely: origin-based, price-based, and availability-based constraints. The time constraint - if event has already happened or not - is a common combination applied to all other constraints. Further, we have described tests regarding royalty payment.

6.2.1.1 Origin-based constraints

Origin-based constraints are the following:

- Revert arbitrary transfer, except from whitelisted marketplace, if event has not happened yet
- Allow arbitrary transfer, if event has already happened

The first origin-based rule is covered on listings 6 and 7. On the other hand, the coverage of the second rule is represented on listings 8, 9, and 10.

The test represented on listing 6 asserts that a ticket transfer is indeed stopped with an "EP: transfers locked" exception if the event has not happened yet. We start by preparing the token allowance for the user and then wrap our transfer attempt with an exception assertion.

```
it('Should revert the ticket transfer to external address since event is still
    active', async () => {
  await _prepareAllowance(BigNumber.from(0), true, thirdSigner)

  await expect(
    event
      .connect(secondSigner)
      ['safeTransferFrom(address,address,uint256)'](secondSignerAddress,
          thirdSignerAddress, ticketId),
  ).to.be.revertedWith('EP: transfers locked')
})
```

Listing 6: Should revert the ticket transfer to external address since event is still active

Listing 7 makes sure no third party marketplaces can resell tickets if the event has not happened yet. We have cloned our marketplace and attempted to perform a ticket transfer. An "EP: transfers locked" exception is to be raised.

```
it('Should revert third party marketplaces from reselling tickets if event is
    on-going', async () => {
  thirdPartyMarket = (await marketplaceFixture(
    1,
    yacoobaAdminAddress,
    forwarder.address,
    MarketPlaceType.FixedPrice,
  )) as FixedPrice
  await expect(
    thirdPartyMarket
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress,
          ticketResalePrice, erc20.address),
  )
    .to.emit(thirdPartyMarket, 'ListingCreated')
    .withArgs(0, event.address, ticketId, 0, secondSignerAddress, ticketResalePrice,
        erc20.address, true)

  await event.connect(secondSigner).approve(thirdPartyMarket.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(thirdPartyMarket.address)

  await _prepareAllowance(ticketResalePrice, true, thirdSigner,
      thirdPartyMarket.address)

  await expect(thirdPartyMarket.connect(thirdSigner).buyAssetFromListing(0))
  .to.be.revertedWith(
    'EP: transfers locked',
  )
})
```

Listing 7: Should revert third party marketplaces from reselling tickets if event is on-going

Conversely to listing 6, on listing 8 we want to assess if ticket transfer is possible, after the event occurs. The only difference is that we make use of a helper function, skipToEventEnd, to advance the necessary time.

```
it('Should transfer ticket to an external address since event has already finished',
    async () => {
  await skipToEventEnd()

  await _prepareAllowance(BigNumber.from(0), true, thirdSigner)

  await event
    .connect(secondSigner)
    ['safeTransferFrom(address,address,uint256)'](secondSignerAddress,
        thirdSignerAddress, ticketId)
  expect(await event.ownerOf(ticketId)).to.equal(thirdSignerAddress)
})
```

Listing 8: Should transfer ticket to an external address since event has already finished

Opposite of listing 7, listing 9 makes sure third party marketplaces can resell tickets if the event has already happened.

```
it('Should allow a third party marketplace to resell if event has already finished',
    async () => {
  thirdPartyMarket = (await marketplaceFixture(
  1,
  yacoobaAdminAddress,
  forwarder.address,
  MarketPlaceType.FixedPrice,
 )) as FixedPrice
 await expect(
  thirdPartyMarket
   .connect(secondSigner)
   .createListing(event.address, ticketId, 0, secondSignerAddress,
     ticketResalePrice, erc20.address),
 )
  .to.emit(thirdPartyMarket, 'ListingCreated')
  .withArgs(0, event.address, ticketId, 0, secondSignerAddress, ticketResalePrice,
    erc20.address, true)

 await event.connect(secondSigner).approve(thirdPartyMarket.address, ticketId)
 expect(await event.getApproved(ticketId)).to.eq(thirdPartyMarket.address)

 await skipToEventEnd()

 await _prepareAllowance(ticketResalePrice, true, thirdSigner,
   thirdPartyMarket.address)

 await expect(thirdPartyMarket.connect(thirdSigner).buyAssetFromListing(0))
  .to.emit(thirdPartyMarket, 'ListingSold')
  .withArgs(0, thirdSignerAddress)
 expect(await event.ownerOf(ticketId)).to.eq(thirdSignerAddress)
})
```

Listing 9: Should allow a third party marketplace to resell if event has already finished

Listing 10 represents an important test. It verifies that, if the event has already happened, an arbitrary transfer can be performed using the transferFrom method.

```
it('Should transfer ticket to an external address since event has already finished,
   using transferFrom', async () => {
 await skipToEventEnd()

 await _prepareAllowance(BigNumber.from(0), true, thirdSigner)

 await event.connect(secondSigner).transferFrom(secondSignerAddress,
   thirdSignerAddress, ticketId)

 expect(await event.ownerOf(ticketId)).to.equal(thirdSignerAddress)
})
```

Listing 10: Should transfer ticket to an external address since event has already finished, using transferFrom

6.2.1.2 Price-based constraints

Price-based constraints are the following:

– Revert transfer from whitelisted marketplace, if event has not happened yet and the price is invalid

– Allow transfer from whitelisted marketplace, if event has not happened yet and the price is valid

– Allow transfer at any listing price, if event has already happened

The aforementioned rules are covered by listings 11, 12, and 13, respectively.

On listing 11 an excessive price - one that would exceed the event producer predefined margin - is deliberately provided. We assert that buying an asset listed on the marketplace with a higher than allowed price, before the event happened, raises the "EP: Invalid price" exception.

```
it('Should revert ticket transfer coming from the marketplace if listing price is not
    valid, before the event has finished', async () => {
  listingCount++
  const wrongListingPrice = listingPrice.add('100000')
  await expect(
    marketplace
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress,
          wrongListingPrice, erc20.address),
  )
    .to.emit(marketplace, 'ListingCreated')
    .withArgs(listingCount, event.address, ticketId, 0, secondSignerAddress,
        wrongListingPrice, erc20.address, true)

  await event.connect(secondSigner).approve(marketplace.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(marketplace.address)

  await _prepareAllowance(wrongListingPrice, true, thirdSigner)

  await expect(marketplace.connect(thirdSigner).buyAssetFromListing(listingCount))
  .to.be.revertedWith(
    'EP: Invalid price',
  )
})
```

Listing 11: Should revert ticket transfer coming from the marketplace if listing price is not valid, before the event has finished

The code below, on listing 12, verifies that a ticket transfer with a valid price can occur through the trusted marketplace, before the event happened.

```
it('Should resell ticket if the listing price is valid, before event has finished',
    async () => {
  await expect(
    marketplace
```

```
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress, listingPrice,
          erc20.address),
  )
    .to.emit(marketplace, 'ListingCreated')
    .withArgs(listingCount, event.address, ticketId, 0, secondSignerAddress,
        listingPrice, erc20.address, true)

  await event.connect(secondSigner).approve(marketplace.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(marketplace.address)

  await _prepareAllowance(listingPrice, true, thirdSigner)

  await expect(marketplace.connect(thirdSigner).buyAssetFromListing(listingCount))
    .to.emit(marketplace, 'ListingSold')
    .withArgs(listingCount, thirdSignerAddress)
  expect(await event.ownerOf(ticketId)).to.eq(thirdSignerAddress)
})
```

Listing 12: Should resell ticket if the listing price is valid, before event has finished

Finally, listing 13 ensures no price restriction is enforced through the marketplace, after the event happened. We deliberately provide a higher price than the margin and we advance time until after the event finishes. We then assert that the transfer occurs successfully.

```
it('Should transfer ticket from the marketplace at any listing price, after event has
    finished', async () => {
  await skipToEventEnd()
  const newListingPrice = listingPrice.add(100)
  listingCount++
  await expect(
    marketplace
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress, newListingPrice,
          erc20.address),
  )
    .to.emit(marketplace, 'ListingCreated')
    .withArgs(listingCount, event.address, ticketId, 0, secondSignerAddress,
        newListingPrice, erc20.address, true)

  await event.connect(secondSigner).approve(marketplace.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(marketplace.address)

  await _prepareAllowance(newListingPrice, true, thirdSigner)

  await expect(marketplace.connect(thirdSigner).buyAssetFromListing(listingCount))
    .to.emit(marketplace, 'ListingSold')
    .withArgs(listingCount, thirdSignerAddress)
  expect(await event.ownerOf(ticketId)).to.eq(thirdSignerAddress)
})
```

Listing 13: Should transfer ticket from the marketplace at any listing price, after event has finished

6.2.1.3 Availability-based constraints

Availability-based constraints are the following:

- Revert transfer from whitelisted marketplace, if marketplace is disabled and event has not happened yet
- Allow transfer from whitelisted marketplace, if marketplace is disabled and event has already happened

The availability-based rules are covered by listings 14 and 15, respectively.

The test on listing 14 asserts that a "EP: transfers locked" exception is raised when a marketplace transfer is attempted and the marketplace is disabled, before the event has happened. We have disabled the marketplace to construct the testing scenario.

```
it('Should block transfer if marketplace is disabled, before event has finished',
    async function () {
  expect(await event.connect(yacoobaAdmin).setIsMarketplaceAvailable(false))
    .to.emit(event, 'MarketplaceAvailabilityUpdated')
    .withArgs(false, false)
  listingCount++
  await expect(
    marketplace
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress, listingPrice,
          erc20.address),
  )
    .to.emit(marketplace, 'ListingCreated')
    .withArgs(listingCount, event.address, ticketId, 0, secondSignerAddress,
        listingPrice, erc20.address, true)

  await event.connect(secondSigner).approve(marketplace.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(marketplace.address)

  await _prepareAllowance(listingPrice, true, thirdSigner)

  await expect(marketplace.connect(thirdSigner).buyAssetFromListing(listingCount))
  .to.be.revertedWith(
    'EP: transfers locked',
  )
})
```

Listing 14: Should block transfer if marketplace is disabled, before event has finished

Opposite of the listing above, listing 15 verifies that even with a disabled marketplace, trading can occur after the events ends. The difference in context is that we have advanced time until after the event ends.

```
it('Should allow transfer even if marketplace is disabled, after event has finished',
    async function () {
  expect(await event.connect(yacoobaAdmin).setIsMarketplaceAvailable(false))
    .to.emit(event, 'MarketplaceAvailabilityUpdated')
    .withArgs(false, false)

  await skipToEventEnd()
  listingCount++
  await expect(
    marketplace
      .connect(secondSigner)
      .createListing(event.address, ticketId, 0, secondSignerAddress, listingPrice,
          erc20.address),
  )
    .to.emit(marketplace, 'ListingCreated')
    .withArgs(listingCount, event.address, ticketId, 0, secondSignerAddress,
        listingPrice, erc20.address, true)

  await event.connect(secondSigner).approve(marketplace.address, ticketId)
  expect(await event.getApproved(ticketId)).to.eq(marketplace.address)

  await _prepareAllowance(listingPrice, true, thirdSigner)

  await expect(marketplace.connect(thirdSigner).buyAssetFromListing(listingCount))
    .to.emit(marketplace, 'ListingSold')
    .withArgs(listingCount, thirdSignerAddress)
  expect(await event.ownerOf(ticketId)).to.eq(thirdSignerAddress)
})
```

Listing 15: Should allow transfer even if marketplace is disabled, after event has finished

6.2.1.4 Royalty payment

The following test, shown on listing 16, assesses that no royalties are paid if there are none registered on the event.

```
it('Should transfer without paying royalty since event never registered royalties',
    async function () {
  await skipToEventEnd()

  // by removing the royalty we simulate the event never registered it in the first
      place
  await expect(royaltyRegistry.connect(yacoobaAdmin).removeRoyalty(event.address))
    .to.emit(royaltyRegistry, 'RoyaltyRemoved')
    .withArgs(event.address)

  await _prepareAllowance(BigNumber.from(0), true, thirdSigner, undefined, false)
```

```
    await event
      .connect(secondSigner)
      ['safeTransferFrom(address,address,uint256)'](secondSignerAddress,
          thirdSignerAddress, ticketId)
    expect(await event.ownerOf(ticketId)).to.equal(thirdSignerAddress)
  })
```

Listing 16: Should transfer without paying royalty since event never registered royalties

6.2.2 Marketplace margin

Regarding the marketplace margin, we want to verify the following:

- Only authorized users can change the margin
- The margin cannot be higher than 100%

For the access control verification, we have defined two tests, represented on listings 17 and 18. The first, on listing 17, asserts that users with permissions can indeed change the marketplace margin. To complement, listing 18 asserts that unauthorized users cannot change the marketplace margin.

```
  it('Should change marketplace margin if its the producer or admin of the event',
      async () => {
    for await (const account of [eventProducer, yacoobaAdmin]) {
      const marketplaceMargin = await event.marketplaceMargin()
      await expect(event.connect(account).setMarketplaceMargin(2000))
        .to.emit(event, 'MarketplaceMarginUpdated')
        .withArgs(marketplaceMargin, 2000)
    }
  })
```

Listing 17: Should change marketplace margin if its the producer or admin of the event

```
  it('Should revert if not event producer that tries to change marketplace margin',
      async () => {
    for await (const account of [eventManager, secondSigner]) {
      await expect(event.connect(account).setMarketplaceMargin(2000)).to.be.revertedWith(
        accessControlRevertMessage(await account.getAddress(), producerRole),
      )
    }
  })
```

Listing 18: Should revert if not event producer that tries to change marketplace margin

Lastly, listing 19 verifies that the margin threshold is correctly enforced. The margin cannot be bigger than 10000 (100%). If that is the case, an "EP: margin will exceed sale price" exception should be raised.

```
  it('Should revert if new margin is greater than 10000', async () => {
```

```
  for await (const account of [eventProducer, yacoobaAdmin]) {
    await
        expect(event.connect(account).setMarketplaceMargin(10001)).to.be.revertedWith(
      ’EP: margin will exceed sale price’,
    )
  }
})
```

Listing 19: Should revert if new margin is greater than 10000

6.3 Validation

From our point of view, considering the context of what was developed throughout this project, testing and validation occurred simultaneously. Development steps were immediately followed by tests and peer reviews, in a diligent and continuous way. We believe the synergy of both errands is capable of validating the code is clean and performant while neatly delivering the business requirements. In addition, as we only worked on an individual piece of the system, which is not a user-facing service, further validation from our side was not possible.

6.4 Summary

In this section we started by explaining our testing approach. We’ve carefully looked through the development and testing environment, where Hardhat was augmented with four plugins: Ethers, Typechain, Mocha, and Chai. Basic testing concepts/operations about these tools were presented for initial understanding. This was followed by detailing and exposing our tests. Finally, we left a key note about the validation phase of this project.

7 Conclusion

This master's project aimed at delivering a peer-to-peer decentralized and autonomous protocol which could mitigate the most concerning ticketing industry hitches. The project was implemented for Yacooba, a local startup. The foundation of the protocol was Ethereum, a general-purpose Turing-complete state machine underpinned by a decentralized network of nodes.

We have diligently explored blockchain, Ethereum, and NFTs, in chapter 1. Following that, we have presented the concept of the project, with focus on the design of the protocol, in chapter 2. An extensive research took part in chapter 3 tackling security and optimization techniques to be considered when writing Solidity code for Ethereum. With the same level of interest, we have described approaches to scale blockchains. Lastly, we have stated how royalties can be handled on blockchain protocols. In chapter 4, we have explored related work. Moving to chapter 5, we have explained the business requirements and the development which took progress to achieve them. Lastly, we have reported the testing and validation mechanisms employed to certify the development matched expectations, in chapter 6.

The related work herein provided suggests proposals to mitigate ticket scalping. However, they effectively fail do to so in a trustless way. The first idea refers to a centralized exchange which randomly assigns tickets in the primary market [3]. The ticket transfer is only valid if done through the exchange. The main downside of this solution is the centralization itself. There has to be trust on a sovereign entity. On the other hand, our solution, once deployed, does not require continuous trust since it would live on a decentralized and autonomous system. The rules applied are the ones purely defined on the code. The second proposal introduces batch assignments, with the possibility of randomness, after allocation periods [2]. However, it has the same constraint as the one before - it is controlled by an entity. In fact, this shall be the major pitfall of previous proposed solutions.

Regarding the Ethereum based solutions [52] [53], the main critique applies to both of them. They tried to enforce ticket scalping by providing methods to buy/sell tickets which comply with ticket price definition, but did not think about blocking the possibility of the transfer being performed externally to their methods. For example, they do not override the method safeTransferFrom, inherent in the ERC721 standard, which means the owner of the ticket (or an authorized third-party) could simply connect to the contract and issue a safeTransferFrom, effectively bypassing the existing rules. Therefore, both solutions failed at mitigating ticket scalping. Conversely, we were meticulous enough to think about those edge cases and did override both transferFrom and safeTransferFrom methods. What is more, our trading rules are employed precisely on those methods, suppressing any idea of escape.

Although the protocol notably serves its purpose, there is one clear limitation, a central authoritative marketplace. The NFT ticket is not aware of how much the buyer is paying for it, and therefore it is not possible to enforce price-based rules upon a transfer. For that reason, we only trust on a specific marketplace which we know will be honest and provide us the correct price payed for the transfer.

This project addressed state-of-the-art technologies, namely blockchain, Ethereum, and NFTs. It also showcased best practices and strategies to use them in a secure and efficient way. It coined a protocol which is capable of dealing with long-lived ticketing industry hitches. We therefore believe the project herein implemented can serve as guidance to onboard new users to these technologies, or serve as inspiration for newborn blockchain ideas.

Finally, it can be stated that this implementation has already successfully operated on a real use case, namely the Madeira Blockchain event on November 30, 2022.

7.1 Future work

In terms of the protocol, we should keep researching and being aware of upcoming changes to the EVM and to the Solidity programming language, to keep execution low-cost and secure. Similarly, we should be on the look for a better way to access the price paid for an NFT transfer. We should aim to remove the tight dependency that currently exists between our NFT tickets and an authoritative tailored marketplace. A review by external security auditors should also take place before new releases, reinforcing the safety of the protocol. In addition, and mainly for the same reason, the protocol should be made open-source. Lastly, it should be deployed making use of a layer 2 solution that makes sense at the time, thus reducing costs for both the deployer and protocol users.

References

[1] J. J. Atkinson, “The economics of ticket scalping,” mimeo, Tech. Rep., 2004.

[2] R. Hakimov, C. Heller, D. Kubler, M. Kurino *et al.*, “How to avoid black markets for appointments with online booking systems,” *American Economic Review*, vol. 111, no. 7, pp. 2127–51, 2021.

[3] P. Courty, “Ticket resale, bots, and the fair price ticketing curse,” *Journal of Cultural Economics*, vol. 43, no. 3, pp. 345–363, 2019.

[4] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain technology overview,” *arXiv preprint arXiv:1906.11078*, 2019.

[5] M. Pilkington, “Blockchain technology: principles and applications,” in *Research handbook on digital transformations.* Edward Elgar Publishing, 2016.

[6] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, “Blockchain challenges and opportunities: A survey,” *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

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

[8] V. Buterin, “On public and private blockchains,” accessed on 10-Nov-2021. [Online]. Available: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

[9] M. Liu, K. Wu, and J. J. Xu, “How will blockchain technology impact auditing and accounting: Permissionless versus permissioned blockchain,” *Current Issues in Auditing*, vol. 13, no. 2, pp. A19–A29, 2019.

[10] A. Gauba, “Finality in blockchain consensus,” Aug 2018, accessed on 15-Dec-2021. [Online]. Available: https://medium.com/mechanism-labs/finality-in-blockchain-consensus-d1f83c120a9a

[11] S. Zhang and J.-H. Lee, “Analysis of the main consensus protocols of blockchain,” *ICT express*, vol. 6, no. 2, pp. 93–97, 2020.

[12] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, “On the security and performance of proof of work blockchains,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[13] “The merge,” accessed on 19-Jan-2022. [Online]. Available: https://ethereum.org/en/eth2/merge/

[14] K. Christidis and M. Devetsikiotis, “Blockchains and smart contracts for the internet of things,” *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[15] J. Zahnentferner, “Chimeric ledgers: translating and unifying utxo-based and account-based cryptocurrencies,” *Cryptology ePrint Archive*, 2018.

[16] “Understanding the extended utxo model.” [Online]. Available: https://docs.cardano.org/learn/eutxo-explainer

[17] J. Clifford, “Intro to blockchain: Utxo vs account based,” Sep 2019. [Online]. Available: https://jcliff.medium.com/intro-to-blockchain-utxo-vs-account-based-89b9a01cd4f5

[18] M. Szydlo, “Merkle tree traversal in log space and time,” in *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2004, pp. 541–554.

[19] E. Org, “Patricia merkle trees.” [Online]. Available: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/

[20] V. Saini, “Getting deep into ethereum: How data is stored in ethereum?” [Online]. Available: https://hackernoon.com/getting-deep-into-ethereum-how-data-is-stored-in-ethereum-e3f669d96033

[21] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[22] Joshua, J. Vianello, Hugo, V. Barda, P. Grimaud, E. Yılmaz, V. Garske, K. Moen, and S. Richards, “Erc-721 non-fungible token standard,” accessed on 16-Dec-2021. [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-721/

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

[24] A. Fowler and J. Pirker, “Tokenfication - the potential of non-fungible tokens (NFT) for game development,” in *Extended Abstracts of the 2021 Annual Symposium on Computer-Human Interaction in Play.* ACM, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3450337.3483501

[25] R. Sandford, L. Siri, Dror, Tirosh, Y. Weiss, A. Forshtat, H. Croubois, S. Tomar, P. McCorry, N. Venturo, and F. Vogelsteller, “Eip-2771: Secure protocol for native meta transactions,” Jul 2020, accessed on 06-Aug-2022. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2771

[26] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, “Smart contract development: Challenges and opportunities,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[27] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[28] S. Sayeed, H. Marco-Gisbert, and T. Caira, “Smart contract: Attacks and protections,” *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[29] “Security considerations.” [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/security-considerations.html

[30] S. Team, “Solidity 0.8.0 release announcement,” Dec 2020. [Online]. Available: https://blog.soliditylang.org/2020/12/16/solidity-v0.8.0-release-announcement/

[31] s.-b.-e. solidity-by example, 2022. [Online]. Available: https://solidity-by-example.org/hacks/delegatecall/

[32] C. Diligence, “External calls,” 2021. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/

[33] C. Coverdale, "Solidity: Transaction-ordering attacks," Oct 2020. [Online]. Available: https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e

[34] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2020, pp. 23–34.

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

[36] M. Gupta, "Solidity gas optimization tips," Feb 2019. [Online]. Available: https://mudit.blog/solidity-gas-optimization-tips/

[37] E. Organization, "Layer 2," 2021. [Online]. Available: https://ethereum.org/en/layer-2/

[38] MongoDB, "Database sharding: Concepts amp; examples." [Online]. Available: https://www.mongodb.com/features/database-sharding-explained

[39] E. Organization, "Sharding." [Online]. Available: https://ethereum.org/en/upgrades/sharding/

[40] C. Staff, "Blockchain technology: Layer-1 and layer-2 networks," Mar 2022. [Online]. Available: https://www.gemini.com/cryptopedia/blockchain-layer-2-network-layer-1-network#section-layer-2-scaling-solutions

[41] district0x, "The basics of state channels," Oct 2018. [Online]. Available: https://education.district0x.io/general-topics/understanding-ethereum/basics-state-channels/

[42] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols," in *International Conference on Financial Cryptography and Data Security.* Springer, 2020, pp. 201–226.

[43] IvanOnTech, "Layer 2 explained - what are layer-2 solutions?" Oct 2021. [Online]. Available: https://academy.moralis.io/blog/layer-2-explained-what-are-layer-2-solutions

[44] wackerow, corwintines, minimalsm, e. awosika, hursittarcan, samajammin, jillweldon, D. Nolan, and fulldecent, "Plasma chains." [Online]. Available: https://ethereum.org/en/developers/docs/scaling/plasma/

[45] D. W. Makers, "Coding bootcamps in virginia." [Online]. Available: https://www.coding-bootcamps.com/blog/how-plasma-chains-work-in-ethereum.html

[46] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, vol. 72, pp. 201–224, 2014.

[47] "Architecture: Polygon technology: Documentation," Oct 2022. [Online]. Available: https://wiki.polygon.technology/docs/maintain/validator/architecture

[48] wackerow, corwintines, minimalsm, e. awosika, hursittarcan, samajammin, jillweldon, D. Nolan, and fulldecent, "Scaling." [Online]. Available: https://ethereum.org/en/developers/docs/scaling

[49] “Optimistic rollups.” [Online]. Available: https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/

[50] “Zero-knowledge rollups.” [Online]. Available: https://ethereum.org/en/developers/docs/scaling/zk-rollups/

[51] Z. Burks, J. Morgan, B. Malone, and J. Seibel, “Eip-2981: Nft royalty standard,” Sep 2020. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2981

[52] P. Corsi, G. Lagorio, and M. Ribaudo, “Ticketh, a ticketing system built on ethereum,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 409–416.

[53] F. Regner, N. Urbach, and A. Schweizer, “Nfts in practice–non-fungible tokens as core component of a blockchain-based event ticketing application,” 2019.

[54] S. Pittet, “The different types of testing in software.” [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

[55] “Ethers: Ethereum development environment for professionals by nomic foundation.” [Online]. Available: https://hardhat.org/hardhat-runner/plugins/nomiclabs-hardhat-ethers#helpers

[56] “Wallets and signers¶.” [Online]. Available: https://docs.ethers.org/v4/api-wallet.html

[57] “Hardhat chai matchers: Ethereum development environment for professionals by nomic foundation.” [Online]. Available: https://hardhat.org/hardhat-runner/plugins/nomicfoundation-hardhat-chai-matchers

[58] “Reference: Ethereum development environment for professionals by nomic foundation.” [Online]. Available: https://hardhat.org/hardhat-network/docs/reference#hardhat_mine

a

Complete unit test list

Context	Test case
Event	Should get current event implementation and default protocol fee
Event	Should revert to create a new event with insufficient allowance
Event	Should create an event through event factory
Event	Should revert a new initialization of the event
Event	Should revert if ether is sent directly to the event contract
Event	Should have the implementation empty
EventsFactory	Should revert event Factory when margin is greater than 10000
EventsFactory	Should revert event Factory when producer address is set to zero address
EventsFactory	Should revert event Factory when no producer address is set
EventsFactory	Should initialize event Factory when producer address is set but includes zero address
EventsFactory	Should have the event Factory correctly initialized
EventsFactory	Should revert the creation of event Factory when ERC20 Address is zero
EventsFactory	Should revert event the creation of Factory when event creation fee is zero
EventsFactory	Should revert if not owner tries to update creation fee
EventsFactory	Should revert if owner tries to set creation fee erc20 address to zero
EventsFactory	Should revert if owner tries to set creation fee to zero
EventsFactory	Should create a new event, emit event and have correct properties
EventsFactory	Should update the admin if called by current admin
EventsFactory	Should fail to update the admin if not called by current admin
EventsFactory	Should update the current fee if called by current admin
EventsFactory	Should fail to update fee if not called by current admin
EventsFactory	Should fail to update fee if value is greater that 10000
EventsFactory	Should update the event implementation
EventsFactory	Should fail to update the event implementation if not called by current admin

Forwarder	Should have the correct ForwardRequest type
Forwarder	Should verify correctly a signature
Forwarder	Should forward a call and correctly execute it
Forwarder	Should fail when domain separator is wrong
Forwarder	Should fail to forward a call when the message is not properly signed (req.from != signature)
Forwarder	Should fail when the nonce is incorrect
Forwarder	Should revert forward a call when not the owner executing
Forwarder	Should batch transactions from same sender, execute and revert due a bad call
Forwarder	Should batch transactions from same sender and execute
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and execute
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and execute even if one tx reverts
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and revert if one fails
Forwarder	Should fail if ether is sent directly to forwarder contract
Forwarder	Should verify correctly a signature
Forwarder	Should forward a call and correctly execute it
Forwarder	Should fail when domain separator is wrong
Forwarder	Should fail to forward a call when the message is not properly signed (req.from != signature)
Forwarder	Should fail when the nonce is incorrect
Forwarder	Should revert forward a call when not the owner executing
Forwarder	Should batch transactions from same sender, execute and revert due a bad call
Forwarder	Should batch transactions from same sender and execute
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and execute
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and execute even if one tx reverts
Forwarder	Should batch several requests/txs in independent calls (diff reqs) and revert if one fails
Forwarder	Should fail if ether is sent directly to forwarder contract
EventManager	Should set new ticket supply for managers, producers and admin
EventManager	Should revert when caller has no role assigned to call setTicketSupply
EventManager	Should set new ticket supply if event already started
EventManager	Should revert when new ticket supply smaller
EventManager	Should revert if ticket tier is disabled

EventManager	Should set new start date as manager, producer and yacooba admin
EventManager	Should revert if setting a new date if event already started
EventManager	Should revert when caller has no role assigned to call setDate
EventManager	Should revert when new start date is in the past
EventManager	Should set new ticket tier sales date as manager, producer and yacooba admin
EventManager	Should set date for sales to end after event start
EventManager	Should revert if setting new ticket sale date during a sale
EventManager	Should revert when caller has no role assigned
EventManager	Should revert when new start sales date is in the past
EventManager	Should revert when the new date to end sales happens before the start sales date
EventManager	Should revert if ticket tier is disabled
EventManager	Should add a new ticket tier as manager, producer and yacooba admin and get it
EventManager	Should revert to add a new ticket when caller has no role assigned
EventManager	Should add a new ticket tier if event already started
EventManager	Should add a new disabled ticket tier
EventManager	Should revert when adding ticket tier with start date before end date
EventManager	Should revert when adding new ticket tier without ticket supply
EventManager	Should disable a ticket tier as manager, producer and yacooba admin
EventManager	Should revert when caller has no role assigned is disabling ticket tier
EventManager	Should revert when disabling ticket tier that does not exist
EventManager	Should disable marketplace
EventManager	Should not have permission to disable marketplace
EventManager	Should enable marketplace
EventManager	Should not have permission to enable marketplace
EventManager	Should enable marketplace if marketplace margin is set
EventManager	Should pause ticket sales by Yacooba Admin
EventManager	Should revert pausing when caller is not yacooba admin
EventManager	Should pause and unpause ticket sales
EventManager	Should withdraw 10€ from the ticket sales as producer and admin
EventManager	Should perform two consecutive withdraws
EventManager	Should withdraw all the funds

EventManager	Should revert if the amount exceeds the event funds
EventManager	Should revert when caller is a manager
EventManager	Should revert when caller has no role
EventTickets	Should buy a new ticket and provide fee to admin
EventTickets	Should buy a new ticket without being yacooba admin
EventTickets	Should revert if a purchase for one ticket tier fails when buying multiple tickets
EventTickets	Should revert if getting nonexistent ticket
EventTickets	Should revert if token transfer was not approved
EventTickets	Should revert if attendee have no funds
EventTickets	Should revert if sale in not active
EventTickets	Should revert if event is sold out
EventTickets	Should revert if ticket tier is disabled
EventTickets	Should set a new fee by protocol admin
EventTickets	Should fail when set a new fee with value greater than 10000
EventTickets	Should fail when set a new fee by other account other than the protocol admin
EventTickets	Should revert if ether is sent directly to the event contract
EventTickets	Should revert the ticket transfer to external address since event is still active
EventTickets	Should transfer ticket to an external address since event has already finished
EventTickets	Should resell ticket if the listing price is valid, before event has finished
EventTickets	Should revert ticket transfer coming from the marketplace if listing price is not valid, before the event has finished
EventTickets	Should allow transfer even if marketplace is disabled, after event has finished
EventTickets	Should block transfer if marketplace is disabled, before event has finished
EventTickets	Should transfer ticket from the marketplace at any listing price, after event has finished
EventTickets	Should revert third party marketplaces from reselling tickets if event is on-going
EventTickets	Should allow a third party marketplace to resell if event has already finished
EventTickets	Should transfer ticket to an external address since event has already finished, using transferFrom
EventTickets	Should transfer without paying royalty since event never registered royalties
EventTickets	Should change marketplace margin if its the producer or admin of the event
EventTickets	Should revert if new margin is greater than 10000

EventTickets	Should revert if not event producer that tries to change marketplace margin
FixedPrice	Should revert to list an ERC721 asset with paused contract
FixedPrice	Should revert to list an asset with ERC20 address as 0x0
FixedPrice	Should revert to list a ERC721 asset that seller is not the owner
FixedPrice	Should revert to list an ERC1155 asset that seller is not the owner
FixedPrice	Should revert to list a nonexistent ERC721 asset
FixedPrice	Should revert to list a nonexistent ER1155 asset
FixedPrice	Should create a ERC721 listing for an asset
FixedPrice	Should create a listing for an ERC1155 asset
FixedPrice	Should create a new ERC721 listing and increment the listing ID
FixedPrice	Should create a new ERC1155 listing and increment the listing ID
FixedPrice	Should revert to buy an ERC721 asset with paused contract
FixedPrice	Should revert to buy an ERC1155 asset with paused contract
FixedPrice	Should revert if buyer do not allow market to move ERC20 funds
FixedPrice	Should revert if Seller does not autorize the asset transfer or his not owner of the asset anymore
FixedPrice	Should buy an ERC721 asset from the listing
FixedPrice	Should buy an ERC1155 asset from the listing
FixedPrice	Should revert the remove of listing of ERC721 as a random account
FixedPrice	Should revert the remove of listing of ERC1155 as a random account
FixedPrice	Should remove listing of ERC721 as market owner
FixedPrice	Should remove listing of ERC1155 as market owner
FixedPrice	Should remove listing of ERC721 as seller
FixedPrice	Should remove listing of ERC1155 as seller
Auction	Should revert to list an ERC721 asset with a paused contract
Auction	Should revert to list an asset with auction ending in less than 5 blocks
Auction	Should revert to list an asset with ERC20 address as 0x0
Auction	Should revert to list a ERC721 asset that seller is not the owner
Auction	Should revert to list an ERC1155 asset that seller is not the owner

Auction	Should revert to list a nonexistent ERC721 asset
Auction	Should revert to list a nonexistent ER1155 asset
Auction	Should create a ERC721 listing for an asset
Auction	Should create a listing for an ERC1155 asset and increment listing ID
Auction	Should create a listing for asset with staring price of zero
Auction	Should revert the remove of listing of ERC721 from a random seller account
Auction	Should revert the remove of listing of ERC1155 from a random seller account
Auction	Should remove listing of ERC721 as market owner
Auction	Should remove listing of ERC1155 as market owner
Auction	Should remove listing of ERC721 as seller
Auction	Should remove listing of ERC1155 as seller with bid reverted to bidder
Auction	Should revert to bid an ERC721 asset with paused contract
Auction	Should revert to bid an ERC1155 asset with paused contract
Auction	Should revert if buyer do not allow market to move ERC20 funds
Auction	Should revert if Seller does not increase allowance for ERC20
Auction	Should revert when seller tries to bid on it's own auction
Auction	Should bid on ERC721 listing
Auction	Should bid on ERC1155 listing
Auction	Should bid on ERC721 listing with a higher amount
Auction	Should bid on ERC1155 listing with a higher amount
Auction	Should revert the bid if bidder already has the highest bidder
Auction	Should revert if new bid equal than previous bid value
Auction	Should revert if new bid lower than previous bid value
Auction	Should revert if new bid happens after auction ended
Auction	Should revert auction can't be settled before it reaches the end
Auction	Should settle auction without executing any transfer because there are no bidders
Auction	Should settle auction and transfer the item to highest bidder