Secure Abstraction of Fractionalization Smart Contracts for Non-Fungible Tokens

Wejdene Haouari

A Thesis Submitted to the Faculty of Graduate Studies
In Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science

Graduate Program in Computer Science

York University
Toronto, Ontario

January 2024

Abstract

Non-fungible tokens (NFTs) have faced a downturn in interest, prompting a critical reassessment of their utility and accessibility. Fractionalization emerges as a solution, by enabling multiple parties to hold a stake in a single NFT, fractionalization lowers the barrier to entry for investors, enhancing market liquidity.

Implementing fractionalization relies on smart contracts, which govern the terms of division, and transfer of fractions of an NFT. However, the complexity of these contracts and the immutable nature of blockchain underscores the importance of security.

This thesis tackles the challenge of implementing fractionalization solutions and enhancing the security of supporting smart contracts. It thoroughly analyzes current fractionalization methods, identifies security vulnerabilities, and explores mitigation strategies to contribute to a safer and inclusive NFT ecosystem. The goal is to propose a baseline for standardizing NFT fractionalization to improve interoperability and address security concerns, laying the groundwork for a more unified and secure ecosystem.

Keywords: blockchain, fractionalization, non-fungible tokens, Ethereum, smart contracts, Security .

Dedication

It is with genuine gratitude and warm regard that I dedicate this work:

To my dear parents, Leila and Ali, for their infinite love and tireless support. Their unwavering dedication and encouragement have been a constant source of inspiration. I am forever indebted to them for their endless devotion and the immeasurable impact they have had on my life.

To my sister Nour, whose positive influence has inspired me to work hard and strive for my goals. I cannot adequately express my gratitude and admiration for you.

To my dear friends, I am grateful for your endless love and unwavering support.

To my friends Meriem, Emna, Rym, and Ghofrane. Thank you for being my second family abroad and for all the beautiful memories we have created together.

To my professors for their advice and support.

Acknowledgments

At the end of this journey, I would like to express my sincere gratitude to everyone who supported me and helped me to accomplish this Master's program in the best conditions.

I want to express my sincere gratitude to Professor Marios Fokaefs, my supervisor, for his invaluable guidance, unwavering support, and endless patience throughout my Master's program. His profound expertise and insightful perspectives have not only inspired my research but have also opened doors to remarkable opportunities for which I am eternally grateful.

I sincerely thank the esteemed committee members, including Prof. Sotirios Liaskos and Prof. Kostas Kontogiannis, for taking the time to examine and evaluate my dissertation.

I am grateful to Mitacs for providing various opportunities that have significantly improved my academic and professional experience.

Last but not least, I'd like to take this opportunity to thank all of my family members, professors, and friends who have led and supported me throughout this journey.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1:

Introduction

This chapter delves into the core motivations driving our study, outlines the essential research questions we aim to answer, and describes the methodology adopted to navigate these inquiries. It also highlights our contributions to the field.

1.1 Context

In the rapidly evolving landscape of blockchain technology, Non-Fungible Tokens (NFTs) have emerged as a significant innovation [1], transforming how we think about digital ownership and asset management. Despite their initial increase in popularity, the NFT market has recently faced challenges, including a noticeable decline in interest and questions about their long-term viability and accessibility. These challenges underscore the need for innovative solutions to revitalize the market and make NFTs more accessible and appealing to a broader audience. Among these solutions, NFT fractionalization [2] presents a promising solution by enabling shared ownership of digital assets, thereby lowering the entry barrier for potential investors and expanding the market beyond its current confines.

However, the implementation of fractionalization within the NFT ecosystem is not straightforward. It relies heavily on the robustness of Ethereum smart contracts, which encode the rules for buying, transferring, and managing on the blockchain [3]. Security vulnerabilities within these contracts, such as reentrancy attacks, overflow errors, or improper access controls [4], could lead to exploitation scenarios, endangering the assets and compromising trust in the fractionalization

model. Addressing these security challenges through understanding existing vulnerabilities, testing, and adherence to best practices in smart contract development is vital for NFT fractionalization's reliability and safe implementation.

This thesis aims to tackle the challenges associated with the fractionalization of NFTs. It begins by exploring the concept of NFT fractionalization and analyzing existing solutions to propose an abstract model that could serve as a standard baseline. Next, the focus shifts to critically examining Ethereum smart contract security. The work reviews existing vulnerabilities and assesses tools for vulnerability detection to develop mitigation strategies that can strengthen smart contract integrity. These strategies are crucial to address the security concerns that arise when fractionalizing NFTs and ensure the trustworthiness of the fractionalization process. In the final section, the thesis combines the findings from the analysis of fractionalization and smart contract security. It outlines a concrete solution for NFT fractionalization that complies with security protocols, facilitates fractional ownership, and rigorously safeguards against security vulnerabilities.

1.2 Motivation

This thesis is motivated by the urgent need to tackle two significant challenges within the blockchain and NFT landscapes: the effective implementation of NFT fractionalization and enhanced security protocols for Ethereum smart contracts.

Firstly, the concept of NFT fractionalization offers a promising pathway to democratize the ownership of digital assets, making them more accessible to a broader audience. This approach can potentially revitalize the NFT market by lowering the barriers to entry and introducing NFTs to new sectors and demographics. However, The current landscape of NFT fractionalization lacks uniformity, leading to various implementations with differing security and functional qualities. This lack of standardization restricts interoperability between platforms and poses significant security risks.

Secondly, the security of Ethereum smart contracts, which are pivotal for managing fractionalized NFTs, emerges as a critical concern. The immutable nature of blockchain technology means

that any vulnerabilities in smart contracts can have far-reaching and permanent consequences. Despite this, there is a noticeable deficiency in security guidelines for smart contract development, particularly for those new to the field. This gap increases the likelihood of security flaws.

1.3 Research Questions

This thesis provides a thorough examination of NFT fractionalization and smart contract security.

RQ1: How can NFT fractionalization be standardized to enhance uniformity and interoperability?

- **RQ1.1:** What are the existing mechanisms and practices in NFT fractionalization?
- **RQ1.2:** How can an abstracted framework for NFT fractionalization be developed to enhance uniformity, interoperability, and security?

RQ2: How can the security of Ethereum smart contracts be enhanced, and comprehensive guidelines be developed for new developers?

- **RQ2.1:** What are the most common and critical vulnerabilities in current Ethereum smart contracts, and how do they affect the blockchain ecosystem?
- **RQ2.2:** What best practices can be established to mitigate these vulnerabilities in Ethereum smart contracts, ensuring their reliability and security?

RQ3: How can a concrete and secure solution be developed to implement NFT fractionalization, ensuring robustness, reliability, and seamless system interaction?

- **RQ3.1:** What security mechanisms are essential for ensuring the resilience and reliability of fractionalized NFTs?
- **RQ3.2:** What methodologies and testing procedures can be employed to validate the security and the seamless system interaction of the proposed solution for NFT fractionalization?

1.4 Methodology

Our methodology is structured into distinct phases, each targeting specific aspects of Ethereum smart contract security and NFT fractionalization.

Analysis of Popular Fractionalization Solutions

To answer RQ1.1, which is centered around the mechanisms and practices currently used for NFT fractionalization, the study will begin by reviewing reports and articles about NFT fractionalization. Subsequently, the study will thoroughly analyze three popular fractionalization solutions. The focus of this analysis will be on evaluating the functionalities of each solution. By examining these existing solutions, the study aims to gain insights into the current landscape of NFT fractionalization and identify common approaches.

Proposal of an Abstraction for Standardization

To answer RQ1.2, we propose an abstraction that focuses on standardizing the fractionalization process. This phase involves synthesizing the strengths and addressing the limitations observed in existing solutions to develop a more universal and efficient framework for NFT fractionalization. The proposed abstraction is intended to facilitate interoperability between different platforms and enhance the security and functionality of fractionalized NFTs.

Multivocal Literature Review (MLR) for Ethereum Smart Contract Security

To answer RQ2.1, we conducted a Multivocal Literature Review (MLR). This approach was chosen because it effectively synthesizes knowledge from various sources, including academic and grey literature. Our review covers peer-reviewed articles, industry reports, white papers, and blog posts. The MLR aims to understand the current state of Ethereum smart contract security, identify common vulnerabilities, and compile existing solutions and best practices with a specific focus on the impact of NFT fractionalization.

Development of Guidelines for New Developers

To address RQ2.2, we analyzed the results to formulate guidelines for new developers. This step is crucial for translating the theoretical and empirical findings into practical guidance. The guidelines focus on common security issues, coding best practices, and preventive strategies to enhance the security of smart contracts.

Implementation of a Concrete Solution for NFT Fractionalization and Security Assessment

To answer RQ3, we moved on to the practical application of the methodologies and insights we gathered from the previous phases. Firstly, we examined the audit reports of the existing fractionalization solution to identify the vulnerabilities that NFT fractionalization may face. After that, we designed a concrete solution for NFT fractionalization, which incorporated the proposed standardization abstraction and the previously developed security guidelines. We then conducted a thorough security assessment of this solution.

1.5 Contribution

This research has made three key contributions to the field of blockchain technology.

The first contribution is exploring the concept of NFT fractionalization, which involves assessing the security of popular fractionalization solutions and proposing an abstraction for this process. This contribution promotes uniformity, interoperability, and enhanced security in NFT markets, especially within sectors like decentralized finance (DeFi) and tokenization. The findings have been published in a paper titled "Towards a Standardization of Fractionalization Smart Contracts for Non-Fungible Tokens," featured in the proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering (CASCON '23) on pages 132–141, September 2023 [5].

The second contribution is the security analysis of Ethereum smart contracts to provide guidelines for new developers. These guidelines, derived from a Multivocal Literature Review, address

critical security vulnerabilities and best practices, offering a valuable resource to enhance the security and reliability of blockchain applications. The results are published in Arxiv [6], and we have submitted this publication to IEEE Access. However, it was rejected because a similar review was published while ours was under review.

The third contribution is the participation in implementing a benchmarking platform for the blockchain network, which supports three different blockchains, Ethereum, Hyperledger Fabric, and Sawtooth, with a combination of different consensus. It can also be integrated into both public and private blockchain. Two publications were the results of this tool. The first publication, titled "Public or Private?: A Techno-Economic Analysis of Blockchain," is featured in the proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON '21) on pages 83–92, November 2021 [7]. The second publication, "BlockCompass: A Benchmarking Platform for Blockchain Performance," is currently under review in IEEE Transactions on Knowledge and Data Engineering.

1.6 Thesis Organization

The structure of this thesis is outlined as follows. Chapter 2 introduces the foundational aspects of Ethereum smart contracts and NFT fractionalization, setting the stage for understanding the core technologies used in this research. Chapter 3 delves into the concept of fractionalization, discussing both theoretical perspectives and practical applications, and proposes a novel abstraction for fractionalization smart contracts. Chapter 4 presents a detailed examination of smart contract vulnerabilities, mitigation strategies, and their relevance to the NFT and fractionalization domain. Chapter 5 assesses the security mechanisms within existing fractionalization solutions through audit reports and showcases a practical implementation of the proposed standard to evaluate its security features. The thesis concludes with a summary of the findings and suggestions for future research directions.

Chapter 2:

Background and Related Work

This chapter presents the background information and related work pertinent to Ethereum smart contracts and NFT fractionalization. It lays the foundation for understanding the current state of the art and situates the research within the broader context of blockchain technology.

2.1 Background

Blockchain Technology

A blockchain [8] is fundamentally a digital ledger comprised of a sequence of blocks, each containing a collection of transactions or digital records. Distinctively, it operates on an append-only basis, meaning that once data is added to a block and the block is appended to the chain, it cannot be altered or removed. This characteristic is visually represented in a structure where each block is cryptographically linked to its predecessor, forming a continuous chain.

At its core, a blockchain maintains a complete and verifiable record of all transactions from the first block, the genesis block, to the most recent. In addition to transaction data, each block typically includes a timestamp and a cryptographic hash of the previous block, adding to the security and integrity of the entire chain. One of blockchain technology's defining features is its decentralized nature. The ledger is not stored in a single location or managed by a central authority; instead, it is distributed across a network of nodes. These nodes collaboratively maintain the blockchain, making it highly resistant to tampering and fraudulent activities.

Security and consensus [9] in a blockchain are maintained through various mechanisms, depending on the specific blockchain. These mechanisms, such as Proof-of-Work (PoW) in Bitcoin and Proof-of-Stake (PoS) in Ethereum 2.0, are critical for validating transactions and ensuring the consistent and reliable addition of new blocks to the blockchain. These protocols are essential for establishing trust and authenticity in the blockchain network, making it a robust and secure platform for digital transactions and record-keeping.

Ethereum

Ethereum is a widely-used open-source platform for decentralized applications, known for its Turing-complete programming languages [10]. This allows for the creation of detailed smart contracts and decentralized apps with custom transaction rules and state changes. Ethereum aims to build a decentralized, censorship-resistant network governed by these user-created smart contracts, eliminating the need for central authorities. The platform's native cryptocurrency, Ether, is used for transaction fees and network services. At Ethereum's core is the Ethereum Virtual Machine (EVM), which runs in an isolated environment on each node, executing smart contract code securely and independently. Ethereum employs a 'gas' system for executing operations, where each action in a smart contract is assigned a gas cost to prevent resource misuse and ensure efficient operation. This system contrasts with Bitcoin's transaction-based charging, focusing instead on the computational effort of each operation.

Ethereum Smart Contracts

Smart contracts [11] on the Ethereum platform are self-executing contracts with the terms of the agreement directly written into lines of code. These contracts exist across a distributed, decentralized blockchain network. Unlike regular user accounts on Ethereum, which individuals control, smart contracts are autonomous and operate based on pre-defined rules set in their code. This code governs the conditions under which the contract is executed, ensuring that agreements are upheld without needing a central authority or intermediary.

Each smart contract on Ethereum has its own internal state where it can store data, ranging

from cryptocurrency balances to complex data structures. This feature allows smart contracts to function in diverse roles, from managing cryptocurrency transactions to running decentralized applications (dApps).

Smart contracts are typically written in high-level programming languages like Solidity or Vyper [12], which are then compiled into Ethereum Virtual Machine (EVM) bytecode. This bytecode is executed when the contract receives a message or transaction from a user or another contract. During execution, the contract can perform operations such as reading from and writing to its internal storage, transferring Ether, and interacting with other contracts and users. Smart contracts ensure that the terms of the contract are executed reliably and transparently. A contract's entire state and transaction history is publicly visible on the blockchain, which allows for verification and auditing of its operations. However, this does not inherently guarantee the security or reliability of the contract's code.

Solidity

Solidity is a statically typed programming language [13] designed for writing smart contracts on the Ethereum blockchain. Solidity is dedicated to the Ethereum Virtual Machine (EVM) and includes specific functionalities like state variables, function modifiers, and events for smart contract development. A key aspect of Solidity is its handling of opcodes, the low-level operations executed on the EVM. These opcodes are crucial for defining the contract's behavior and interaction with the blockchain. Security is a concern in Solidity due to the immutable nature of blockchain. The language incorporates features to mitigate common vulnerabilities and attacks. Still, the irreversible nature of smart contracts on Ethereum makes it critical for developers to test and audit their code to prevent security breaches rigorously.

Gas and Fees

Gas on the Ethereum network symbolizes the computational resources needed to execute an operation or a series of operations (such as a transaction or smart contract execution) [14]. Each operation is assigned a gas cost, reflecting its computational complexity. For instance, a simple ETH

transfer requires a gas cost of 21,000 units. In contrast, more complex operations, like executing smart contracts, can require significantly more gas depending on the complexity and computational requirements of the contract.

Fees, paid in Ether (ETH) [14], are the product of gas used and the gas price, which is measured in Gwei (one billionth of an Ether). The gas price is determined by market dynamics, where users can offer higher gas prices for their transactions to be prioritized by miners.

Standardization in Blockchain Technology

In the Ethereum ecosystem, smart contract standards are crucial in defining how contracts should operate and interact. To bring consistency and interoperability among various contracts, the Ethereum community has established a set of standards known as Ethereum Requests for Comments (ERCs). These are part of the broader category of Ethereum Improvement Proposals (EIPs) [15].

ERC-20: Token Standard

ERC-20 is a standard for creating and managing tokens on the Ethereum blockchain [16]. Established by Fabian Vogelsteller, it has been instrumental in the growth of Initial Coin Offerings (ICOs) and decentralized finance (DeFi) applications. ERC-20 tokens are fungible, meaning each token is identical and interchangeable with others, similar to conventional currencies or assets like gold. This standard ensures that tokens adhering to the ERC-20 interface are mutually compatible, facilitating their integration across various wallets, exchanges, and decentralized applications.

ERC-721: Non-Fungible Token Standard

The ERC-721 standard, a cornerstone for non-fungible tokens (NFTs) on Ethereum, provides a framework for the creation, management, and trading of unique digital assets [17]. This standard defines a set of functions and events that enable consistent interaction with NFTs. Functions like `safeTransferFrom` allow for secure transfers of NFTs between addresses, while `approve` grants third-party transfer permissions. Meanwhile, the `balanceOf` function reports the quantity of NFTs held by a particular address. ERC-721's unique feature is the ability to represent individual, non-interchangeable assets, making it ideal for digital collectibles, art, and more.

ERC-1155: Multi Token Standard

Developed by Enjin Radomski et al. [18], ERC-1155 is a versatile smart contract standard capable of managing multiple types of tokens within a single contract. This standard supports fungible (like ERC-20) and non-fungible (like ERC-721) tokens, offering a unified approach to token management. This innovation simplifies the process and enhances efficiency in various applications, including gaming, digital collectibles, and artwork.

ERC-165: Interface Detection Standard

ERC-165 specifies a standard method for Ethereum contracts to announce and identify the interfaces they support [19]. This standard is essential for enhancing interoperability among contracts and applications. It allows contracts to query whether a given contract implements a specific interface using the `supportsInterface` function. This capability is handy for developers creating modular and flexible applications, as it enables them to easily determine the functionalities a contract supports and interact with it accordingly.

Decentralized Finance (DeFi)

Decentralized Finance (DeFi) simplifies finance by using technology to manage financial activities like lending, borrowing, and trading without central authorities. It relies on smart contracts on blockchains, mainly Ethereum, to operate transparently and without intermediaries [20]. Liquidity pools, essential components of DeFi, serve as token reserves within smart contracts, enabling direct exchanges of assets or loans. These pools allow users to earn fees proportionate to their contribution. An example of this mechanism in action is Uniswap [21], where users can swap various tokens based on the supply available in the pool, eliminating the necessity for a direct match between buyers and sellers. Furthermore, DeFi introduces the concept of staking, where users lock up their tokens to support network operations in return for rewards, often contributing to liquidity pools or participating in governance decisions. Additionally, yield farming emerges as a strategy for maximizing returns by reallocating assets across different protocols and engaging in lending or staking activities to earn interest or rewards.

2.2 Related Work

NFTs fractionalization

The academic exploration of fractional NFTs remains limited, with most insights coming from industry reports rather than scholarly research.

Fang from *CoinGecko* [22] explores the concept of fractionalized NFTs, focusing on platforms like Unicly and Niftex. They discuss how fractionalization can enhance accessibility and liquidity in the NFT market. However, the analysis would benefit from updating certain information, adding technical details, and including a security assessment to provide a better understanding of the risks and technical intricacies involved in fractionalizing NFTs.

Vijayakumaran [23] analyzes the legal landscape surrounding fractionalized NFTs (F-NFTs), mainly focusing on security law implications. While their exploration offers valuable insights into regulatory considerations, it notably lacks an in-depth discussion on the technical processes and mechanisms underlying F-NFT fractionalization.

Wonseok et al. [24] explore fractional non-fungible tokens (NFTs), emphasizing their role in democratizing access to high-value digital assets through blockchain technology and tokenization. Their paper provides a foundational overview of tokenization, focusing on the potential and challenges of fractional NFTs. Although the study delves into the mechanisms of creation, management, and existing platforms for fractional NFTs, as well as gas fees, it notably does not examine the smart contracts of each solution in detail, particularly their security mechanisms.

Rudytsia and Bogdanova [25] introduce a UML model for NFT fractionalization focusing on technical framework development for blockchain-based property rights distribution, adhering to the ERC-1155 standard. The paper provides a solution for creating, selling, and managing fractional NFTs. However, it does not address the security implications of their proposed model, lacks a buyout option, and specifies that only non-fractionalized NFTs can be purchased, potentially limiting the practicality and adaptability of their approach in diverse application scenarios.

In addition to tool-focused studies, there is a noticeable gap in the security analysis of NFT fractionalization. Current research primarily highlights the benefits of fractionalization for market access and liquidity but often neglects the security challenges involved.

Security of Ethereum Smart Contracts

Research on the security of Ethereum smart contracts covers a broad range of topics, including the analysis of vulnerabilities, examination of analysis tools, verification approaches, and broader blockchain safety concerns.

Several studies have systematically reviewed Ethereum smart contract vulnerabilities and the tools developed to analyze them. Di Angelo et al. [26] conducted a thorough review of Ethereum smart contract analysis tools, categorizing them based on their availability, maturity level, purpose, and analysis method. Despite its coverage, this survey has become somewhat outdated due to the discontinuation of maintenance for some of the tools mentioned.

Durieux et al. [27] performed an empirical analysis of nine automated analysis tools applied to a large dataset of smart contracts, developing a framework for evaluating these tools. However, their analysis did not extensively cover vulnerabilities or mitigation strategies. Similarly, Kushwaha et al. [28] conducted a systematic review of Ethereum smart contract analysis tools, classifying them into static and dynamic analysis categories and scrutinizing 86 tools. This study delves into methodologies such as taint analysis, symbolic execution, and fuzzing. Still, it lacks consideration of the developer community's preferences and the practical scenarios where each tool would be most effective.

On verification approaches, Harz et al. [29] provided an overview of languages, paradigms, and verification approaches for smart contracts. Their discussion, however, did not deeply engage with specific verification tools or vulnerabilities. Further detailing smart contract vulnerabilities and attacks, Kushwaha et al. [30] present a systematic review of security vulnerabilities in Ethereum blockchain smart contracts. The paper categorizes vulnerabilities into three leading root causes and seventeen sub-causes, providing in-depth insights into twenty-four specific vulnerabilities and their prevention methods, detection, and analysis tools. Despite providing a comparative analysis

of various detection tools, the paper falls short in offering in-depth feedback on the effectiveness and limitations of these tools.

Atzei et al. [31] and Chen et al. [32] each presented detailed accounts of significant vulnerabilities and attacks related to Ethereum smart contracts. While Atzei et al. [31] did not focus on mitigation strategies, Chen et al. [32] discussed a range of Ethereum vulnerabilities and associated attacks but did not address vulnerability detection tools. Zhu et al. [33] examined 11 smart contract vulnerabilities in depth, along with defenses against well-known attacks, including schemes to detect these vulnerabilities.

In a broader context, Li et al. [34] discussed general blockchain safety issues and proposed enhancements. In contrast, Saad et al. [35] investigated attacks on various blockchain platforms, including Ethereum, briefly mentioning mitigation strategies.

This thesis aims to address the gaps identified in the existing literature by providing a review and analysis of smart contract vulnerabilities, the effectiveness of various detection tools, and mitigation strategies. Moreover, this work will explore the security challenges of NFT fractionalization platforms, proposing a secure framework for fractional ownership on Ethereum.

Chapter 3:

Fractionalization Smart Contracts for Non Fungible Tokens

In this chapter, we intend to explore the concept of fractionalization from research and practice perspectives and propose an abstraction for fractionalization smart contracts. A significant portion of this chapter is published in the proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering (CASCON '23) on pages 132–141, September 2023 [5].

3.1 Fractionalization

3.1.1 Definition

NFT fractionalization, often referred to as Fractional NFTs (F-NFTs), is the process of breaking down a single Non-Fungible Token (NFT) into smaller, manageable parts known as *fractions* or *shares* [2]. These fractions allow for the shared ownership of an NFT among multiple individuals, thus democratizing access to otherwise prohibitively expensive assets. Each fraction represents a tokenized stake in the NFT, enabling individual investors or collectors to buy and sell their shares independently. This approach is especially beneficial for high-value NFT assets such as exclusive art, luxury items, or real estate, which typically would be unaffordable for an average individual. Through fractionalization, these assets become financially accessible as investors can purchase just a portion of the NFT at a lower cost. Moreover, the fractionalization of NFTs extends the scope of these digital assets beyond traditional domains, such as digital art, to more tangible sectors

like real estate and luxury goods. It also opens up new investment opportunities and scenarios, including in emerging fields like decentralized finance, thus broadening the potential and reach of NFTs in various markets.

Advantages:

NFT fractionalization offers several benefits [36]. First, it enables NFTs to intersect with decentralized finance (DeFi), allowing them to serve as loan collateral. This opens up new financial avenues within the DeFi space. Secondly, fractionalization aids in price discovery. Analyzing the trading activity of fractional shares on the market makes it possible to ascertain the fair market value of an NFT. As high-value NFTs become accessible to a larger group of investors, liquidity and trading volume increase, forming a market-driven price. This price reflects the market's valuation of the NFT, aiding investors and collectors in determining its fair value. Finally, fractionalization significantly increases accessibility, allowing more people to invest in high-value NFTs that were previously unaffordable.

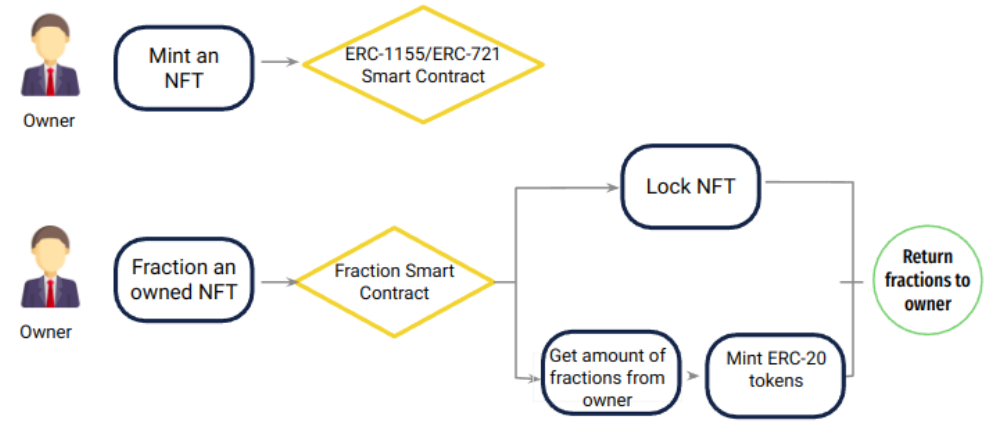
Challenges:

Fractional NFTs also present several challenges [37]. The fractionalization process can be complex, requiring careful definition of ownership terms, legal structures for investment creation, and addressing governance issues. Regulatory uncertainty is another significant challenge, as fractional NFTs inhabit a new and evolving legal area, potentially leading to compliance issues. Additionally, the custody and security of fractional NFTs are critical, demanding secure storage and management to protect investor assets.

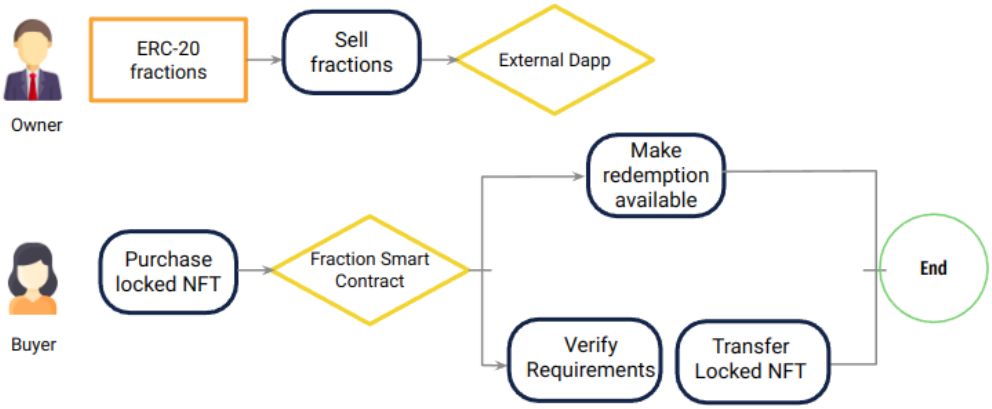
3.1.2 Current Approach

a. Common Process:

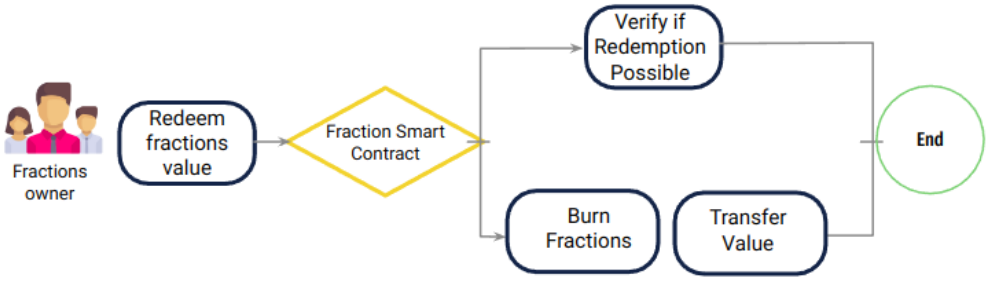
Upon examining existing solutions, we have determined that the fractionalization process can be summarized as depicted in Figure 3.1. The NFT (either an ERC-1155 or an ERC-721 token) is first locked within a smart contract, usually named *vault*, which divides the NFT into multiple fractions. ERC-20 token (fungible tokens) is used to represent fractions. The original NFT owner,



(a) Fraction process



(b) Buyout process



(c) Redemption process

Figure 3.1: How fractionalization works

usually called *curator*, controls several parameters, including the total number of ERC-20 tokens (supply) generated, their pricing, the exit price for NFT redemption, and the metadata used.

After configuring these parameters, the owner can distribute the tokens however they see fit. The most common way of distribution is through a decentralized exchange [38], which allows for seamless trading and higher liquidity for fractionalized NFTs. The smart contract that governs fractionalization typically includes a *buyout* option that will enable users to purchase the locked NFT.

The fraction owners can *redeem* their fractions when the locked NFT is purchased using the buy-out method. This redemption method involves burning the fractions and returning the appropriate value of the fractions from the buyout to the owners.

b. Locking NFTs Within Smart Contracts:

The mechanisms utilized for the secure locking of NFTs are encapsulated in Table 3.1, highlighting the implementation of either the *IERC721Receiver* or *IERC1155Receiver* interface by the vault contract.

The *IERC721Receiver* and *IERC1155Receiver* interfaces are fundamental to enabling smart contracts to reliably accept and manage ERC-721 and ERC-1155 tokens, respectively. They mandate the inclusion of specific functions (*onERC721Received* / *onERC1155Received*) that the recipient contract must implement. These functions are triggered upon the receipt of an NFT, ensuring that the contract is equipped to handle the asset and allowing for the execution of any requisite logic post-receipt [39] [40].

The secure transfer of NFTs into the vault is facilitated through the *safeTransferFrom* method. This method ensures the safe and intended transfer of tokens between addresses by verifying that the recipient address (in this case, the vault contract) implements the necessary receiver interface to manage the incoming NFT. This verification step is crucial, as it prevents the accidental loss of NFTs by ensuring they are not sent to contracts incapable of managing them. When an NFT is successfully transferred to a smart contract address, the corresponding *onERC721Received* / *onERC1155Received* function is invoked to validate and acknowledge the transfer.

Additionally, the vault contract incorporates a mechanism for the controlled release of locked NFTs through the *safeTransfer* method, dependent upon fulfilling predefined conditions. This method is instrumental in ensuring that the transfer of NFTs from the vault adheres to the established security and governance protocols.

Name	**Type**	**Definition**
IERC721Receiver	Interface	Facilitates the reception and management of incoming ERC-721 tokens by a smart contract.
onERC721Received	Function	A requisite function within the *IERC721Receiver* interface for handling the receipt of ERC-721 tokens.
safeTransferFrom	Method	A secure method for transferring ERC-721 or ERC-1155 tokens, ensuring the recipient contract can receive the tokens.
safeTransfer	Method	A method used primarily for ERC-20 token transfers, ensuring the secure movement of tokens between addresses.

Table 3.1: Methods Utilized for the Secure Locking of NFTs

c. Fractionalization methods:

After examining current practices in NFT fractionalization, we identify two primary methodologies: **Aggregation** and **Indexation**, which significantly influence liquidity, market valuation, and investment dynamics.

- **Aggregation Method:** This method involves pooling diverse NFT types (e.g., ERC-1155, ERC-721) into a single collection, allowing investment in fractional stakes. The advantages include the enhancement of accessibility and diversifies investment exposure, reducing risk while providing governance participation to fractional owners. The hallenges include the complexity of extracting specific assets from the pool and the absence of a unified price floor complicating valuation. This method is suitable for investors seeking broad exposure across various NFTs and where governance plays a crucial role in asset appreciation.
- **Indexation Method:** Indexation involves grouping similar NFTs to establish a price floor, simplifying valuation and redemption processes. The advantages include providing market predictability and straightforward exit strategies, establishing a clear benchmark for collection values. However it limits the scope of investment to similar types of NFTs, potentially undervaluing

unique assets. This method is optimal for stable market conditions and precise valuations, appealing to investors focused on specific NFT categories.

d. Buyout mechanisms:

There are several buyout mechanisms for fractionalized NFTs [41], including:

- **Dutch Auction:** In a Dutch auction, the asset's price is gradually lowered until a buyer is found. This mechanism is typically used for time-sensitive assets, incentivizing buyers to act quickly. There are different variants of Dutch auction, such as the last price Dutch auction [42]. Advantages include creating a sense of urgency among buyers, helping discover the asset's actual market value, and providing transparency around the sale process. However, the final price may be lower than the asset's value.
- **Voting Auction:** In a Voting auction, bidders compete to offer the highest price for the asset. This is the traditional auction format and is often used for high-value assets. Advantages include creating a competitive bidding environment to increase the price, allowing buyers to determine the asset's actual market value, and providing transparency around the sale process. However, the auction can be time-consuming and may not attract enough bidders. Moreover, the asset may be sold below its actual value if there is insufficient interest. Also, token holders must actively monitor the buyout bids and vote accordingly.
- **Instant Buyout:** In an instant buyout, a fixed price is set for the asset, and a buyer can purchase the entire asset immediately. This mechanism is often used for low to medium-value assets. Advantages include providing a quick and easy way to buy the entire asset, reducing uncertainty around the sale process, and being suitable for less valuable assets that may not require a more complex sale mechanism. However, the fixed price may not reflect the asset's actual value. It may also not be suitable for high-value assets or assets with complex ownership structures.
- **Buyout Rejection Games:** The process works by a collector buying a portion of the fractional token supply and submitting a bid to purchase the NFTs at a price equal to or greater than the current price. If other speculators or token holders want to cancel the buyout, they can put

money into the system to buy the collector's fractional tokens at the price they submitted when placing their bid. This is done in the hopes that the price of the fractional token will rise in the secondary market. The advantage is that individual token holders will always receive an equal or higher price for their tokens in case of a successful buyout. However, external speculators or token holders who participate in a buyout rejection game by buying the fractional tokens at a higher price to cancel the buyout are taking a risk. There is no guarantee that the price of the NFT in the secondary market (i.e., decentralized exchanges or other platforms) will rise, and they may not be able to sell the fractional tokens for a higher price later on.

e. Redemption:

If a fractional NFT is purchased or sold, token holders can receive a portion of the proceeds based on the percentage of the NFT they own. The mechanism for dispersing the proceeds may differ depending on the platform hosting the fractional NFT and the terms of the investment. The funds may be automatically sent to token holders' platform wallets in some cases; however, if the procedure is done automatically, it results in an enormous amount of gas, making the buyout transaction extremely expensive. Another option is for token holders to initiate a withdrawal or redemption request to collect their part of the sales. The tokens will be burned (destroyed) in this instance, and the value of the tokens will be sent to the fraction holders who invoked the redemption method.

3.2 Current Solutions

3.2.1 Tessera

a. Overview:

Tessera, launched in August 2022, introduces a novel approach to owning and governing Non-Fungible Tokens with significant cultural and historical value, exclusively on the Ethereum blockchain [43]. The platform's foundation is the concept of Hyperstructures [44], which are blockchain protocols designed to operate autonomously, eliminating the need for continuous main-

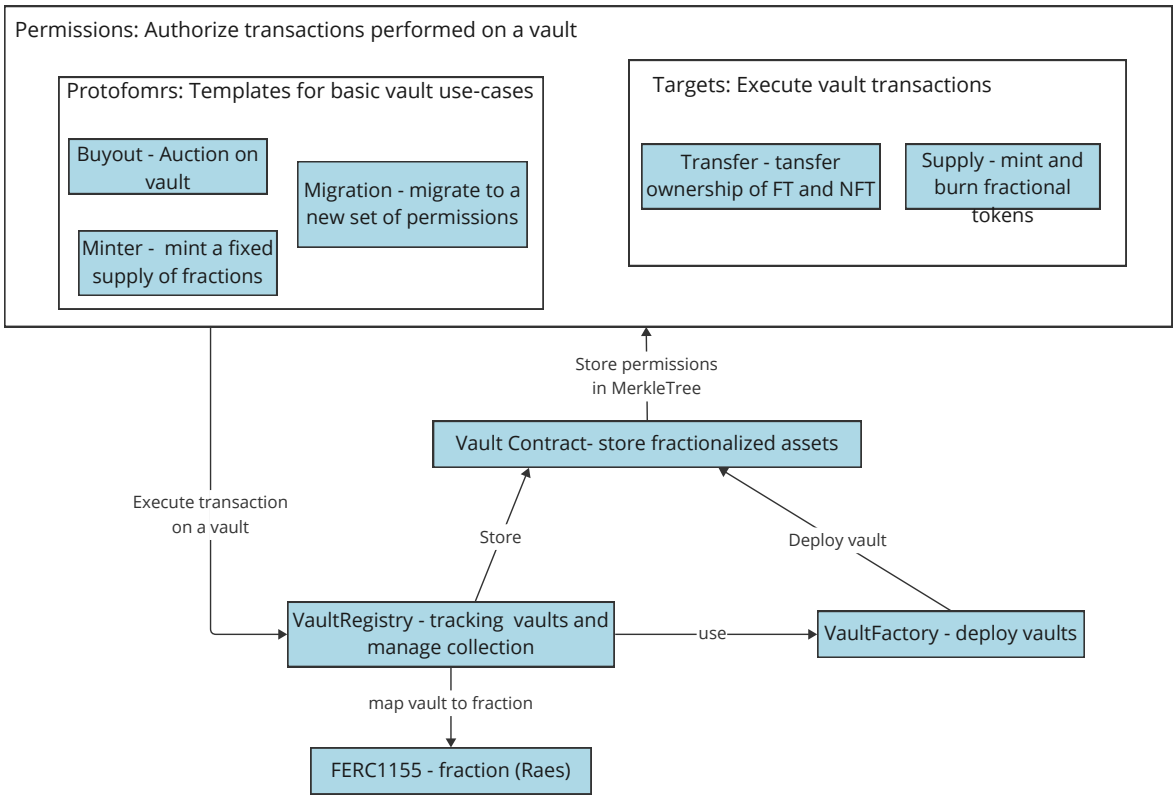


Figure 3.2: Tessera platform smart contracts

tenance, intermediaries, or centralized control. Tessera Platform includes multiple smart contracts; our analysis will concentrate on nine primary contracts [45]. Figure 3.2 shows the interaction between these smart contracts.

At the core of Tessera's functionality is the *Vault Contract*, employing a Merkle Tree-based permission system to ensure security while allowing for post-deployment modifications. Vault creation is facilitated by the *VaultFactory* contract, leveraging cloning technology for efficiency. The *VaultRegistry* maintains a record of vaults and their fractional ownership, represented by *FERC1155* tokens, ensuring transparency and security. Permissions in Tessera are defined by a combination of:

- The *Modules* set of contracts offer advanced functionalities with Protoforms (example smart contracts) for common use cases like *Buyout*, *Minter*, and *Migration*. These functionalities enable auction-based ownership transfers, issuance of tokens with a fixed supply, and seamless transitions

of vaults to new governance models.

- The *Target* contract, simple contracts executed by the vault to manage the lifecycle of tokens, including minting, burning, and transferring, facilitated by the *supply* and *Transfer* contracts.

Although Tessera has taken an innovative approach, it still faces some challenges. One of the main challenges is the lack of integration with decentralized finance (DeFi) platforms. Additionally, using ECR1155 tokens for fractional ownership instead of ERC20 limits its use cases. Although hyperstructures enhance flexibility and programmability, there are also security concerns. One such concern is the ability to alter vault functionalities post-deployment.

b. Smart contracts analysis:

We examined nine core contracts of the Tessera platform, including contracts responsible for the fractionalization process, auctioning, permissions, etc.

1. The `VaultFactory` contract uses the `Create2ClonesWithImmutableArgs` [46] library [46] for efficient deployment of fractional vaults. This is facilitated through the cloning mechanism. The attributes and functions of the smart contract are highlighted in table A.1 and A.2. The contract utilizes the attribute `implementation` to point to a Vault template contract. Noteworthy functions include `deploy()` and `deployFor(address _owner)`, with the latter allowing vaults to be set up for specific owners. They include features to block front-running, which is when someone tries to make a quick profit by acting on information about upcoming trades before those trades happen. The `getNextAddress(address _deployer)` function pre-computes future vault addresses, enhancing predictability and security in vault deployment.
2. The `VaultRegistry` smart contract is designed to manage and track fractional vaults. It utilizes the `ClonesWithImmutableArgs` library as well. The contract handles NFT fraction (fNFT), allowing for the minting and burning of these tokens. Tables A.5 and A.6 summarize the attributes and functions of the smart contract, respectively. Attributes such as `factory`, `fNFT`, and `fNFTImplementation` establish the foundational components of the contract, linking it to essential external contracts for operational functionality. Key functions include `create`, `createFor`, and `mint`, which enable users to establish new vaults with unique permissions and to manage

the fNFTs associated with these vaults.

3. The `Vault` smart contract is designed to manage fractionalized assets. Tables A.3 and A.4 highlight the attributes and functions of the smart contract. `Vault` uses a `MerkleProof` based permission system [47], encapsulated by the `merkleRoot` attribute, to ensure that function calls are authorized. The contract's flexibility is enhanced through a plugin system, where the `methods` mapping allows dynamic function extension via delegate calls to external contracts.

4. The `FERC1155` contract represents fractional NFTs, known as Rea, and enhances the ERC-1155 standard. It introduces scoped approvals, allowing users to grant permission to another address (such as a smart contract or another user) to spend a specific token amount. It uses EIP-712 [48] signatures to grant permission, which enables users to sign off-chain messages. Tables A.7 and A.8 present attributes and functions of the FERC1155 smart contract. Key functions include `mint`, `burn`, and `safeTransferFrom`. The usability is enhanced with `permit` and `permitAll` for off-chain approvals.

5. **Modules smart contracts:**

   A. The `Minter` contract introduces a solution for minting fixed supplies of fractional tokens. Table A.10 summarizes the functions of the smart contract. The `supply` attribute, which is an address pointing to the target `Supply` contract, is responsible for the actual minting process. Through its `getPermissions` function, the contract dynamically generates permissions for minting, encapsulated in a Merkle tree structure. This approach streamlines permission verification. Although internal, the `_mintFractions` function orchestrates the minting by encoding and executing a call to the `supply` contract.

   B. The `Buyout` contract has a mechanism for the buyout of assets within vaults (raes and NFT). Attributes and functions of the smart contract are highlighted in Tables A.11 and A.12. The attributes such as `registry`, `supply`, and `transfer` link the contract to the broader ecosystem by specifying addresses of the *VaultRegistry*, *Supply*, and *Transfer* contracts. The contract has two pivotal periods within a buyout auction: the `PROPOSAL_PERIOD` and the `REJECTION_PERIOD`, defined as constants, which govern the temporal phases of auction proposals. Functionally, the `start` function triggers the initiation of a buyout auction, requiring a deposit of ether and fractional tokens as a bid for vault ownership. This is complemented by the `sellFractions`

and `buyFractions` functions, which allow participants to sell or buy fractional tokens during the auction. The auction's conclusion is managed by the `end` function, determining its success based on the final distribution of fractional ownership. Successful auctions enable asset transfers via the `cash` function, which disburses ether to token holders in proportion to their ownership.

C. The `Migration` contract handles the migration of vaults to a new set of permissions. It interacts with multiple interfaces, including `IERC20`, `IERC721`, `IERC1155`, and `IModule`. Tables A.13 and A.14 present the key attributes and functions of the smart contract. The `registry` and `buyout` attributes link the migration process to *VaultRegistry* and *Buyout* contracts. The contract introduces a `PROPOSAL_PERIOD` constant, defining the time frame for stakeholders to signal support for proposed migrations. The stakeholders participate in the governance process using functions such as `propose`, `join`, and `leave`. The `commit` function is crucial for initiating the process upon reaching target conditions. In contrast, `settleVault` and `settleFractions` functions facilitate the migration of assets and fractional tokens to the new vault.

6. **Targets smart contracts:**

A. The `Supply` contract is the primary mechanism for minting and burning fractional tokens. Tables A.15 and A.16 present the key attributes and functions of the smart contract. The immutable attribute `registry` stores the address of the *VaultRegistry* contract. The contract's core functionalities are embodied in two primary functions: `mint` and `burn`. The `mint` function is designed to issue fractional tokens to a specified address. In parallel, the `burn` function removes a specified amount of fractional tokens from circulation.

B. The `Transfer` contract, is designed to handle the transfer of ERC-20, ERC-721, and ERC-1155 tokens efficiently. It provides three main functions: `ERC20Transfer`, `ERC721TransferFrom`, and `ERC1155TransferFrom` presented in table A.17. Each function is optimized for gas efficiency through the use of assembly code.

3.2.2 Unic.ly

a. Overview:

Unicly, launched in November 2021, is a decentralized NFT system that allows users to tokenize

their assets and engage in trading while including DeFi principles such as staking, yield farming, and liquidity mining [49]. Unicly distinguishes itself through user-friendly design and architecture, making it accessible to a broad audience. Unicly is a platform for manufacturing digital tokens and offering optimal prices for traders, with an NFT creator and an automated market maker (AMM) enabled marketplace [50]. Figure 3.3 illustrates the interaction among the primary smart contracts.

1. uToken Creation and NFT Fractionalization:

- `UnicFactory` initiates the process by minting uTokens, representing fractional ownership of NFTs [51].
- The Converter contract then takes these NFTs, fractionalizes them into uTokens and facilitates auctions for these fractionalized assets. This enables users to buy, sell, or trade fractions of NFTs. The locked NFT can be added to an existing locked collection. To redeem an underlying NFT from the collection, uToken holders must first vote to unlock the collection. Notably, there is no mechanism for bidding on the entire collection; buyers must bid on the underlying NFTs individually. Once the voting threshold is reached, the collection will be unlocked, and the respective highest bidders can claim their NFTs. Subsequently, uToken holders can claim the ETH paid by the bidders.

2. Staking and Earning Rewards:

- `UnicGallery` offers a platform for users to stake their uTokens and earn UNIC tokens as rewards. This staking mechanism incentivizes users to participate in the ecosystem by locking their tokens to support the platform's liquidity and stability.
- Similarly, `UnicFarm` allows users to stake liquidity provider (LP) tokens from various pools to earn UNIC rewards.

3. Governance and Platform Decisions:

- UNIC tokens play a pivotal role in the platform's governance. Holders of UNIC tokens can participate in governance decisions by proposing changes or voting on proposals through the `GovernorAlpha` contract.
- The `Timelock` contract adds a mandatory delay to the execution of governance actions.

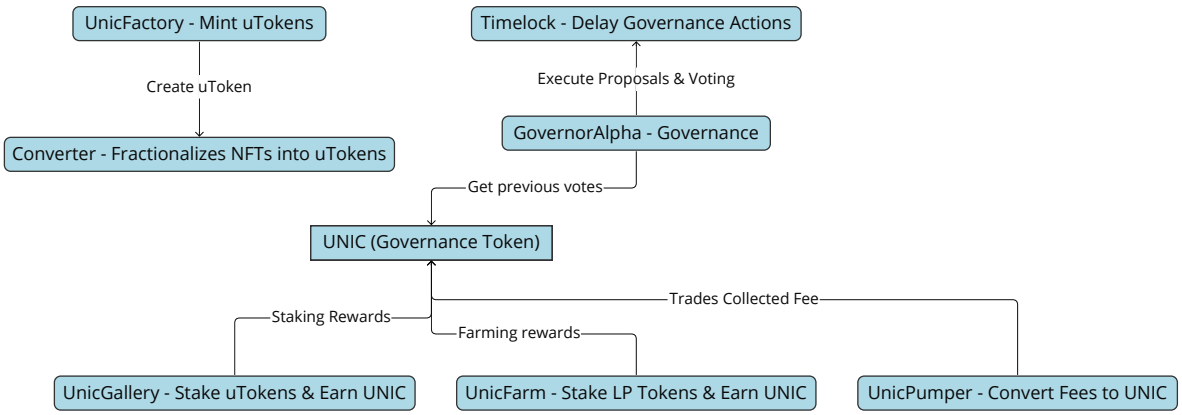


Figure 3.3: Unicly platform smart contracts

4. Fee Conversion and Reward Distribution:

- `UnicPumper` is responsible for converting tokens collected as fees into UNIC tokens. These UNIC tokens are then distributed back into the ecosystem as rewards for staking.

b. Smart contracts analysis:

We examined eight core contracts of the Unicly platform. This includes contracts responsible for fractionalization, governance mechanisms, liquidity provision, and token-swapping functionalities.

1. The `Unic.sol` contract is a governance token within the Unicly ecosystem, combining various ERC standards with governance features. Functioning primarily as a fungible token under the ERC20 standard, it facilitates standard token operations such as transfers and balance checks. Governance aspects of `Unic.sol`, inspired by the Compound protocol [52], allows token holders to delegate voting rights. This mechanism encourages active participation in governance without necessitating the sale or transfer of tokens.

   Tables B.1 and B.2 highlight the attributes and functions in the smart contract. Two technical features of note in `Unic.sol` are *delegateBySig* and optimization functions like *_moveDelegates* and *_writeCheckpoint*. The *delegateBySig* function permits users to delegate their voting power via secure cryptographic signatures, offering a cost-effective and efficient alternative to traditional transactions.

2. The `UnicFactory` is responsible for creating and administrating uTokens. `UnicFactory` attributes and functions can be found in table B.3 and B.4. *createUToken* function is responsible for the generation of uTokens. `FeeTo` is the address for fee collection. Functions like *`setFeeTo`* and *`setFeeToSetter`* ensure that fees are correctly channeled and managed by authorized personnel only. For uToken management, the contract employs a dual system comprising the `uTokens array` and the `getUToken mapping`. The array lists uTokens, while the mapping facilitates quick reference.

3. The `Converter` it combines `ERC20` and `ERC1155Receiver` standards. The contract attributes and functions are presented in table B.5 and B.6. Its main functionalities are the `deposit` and `issue` mechanisms. The `deposit` function enables the contract to accept NFTs adhering to both `ERC721` and `ERC1155` standards. Post-deposit, these NFTs are locked as collateral through the `issue` function, marking the contract's transition from inactive to active.

   The `bid` facilities an auction-style mechanism for NFTs. It employs logic to accept only higher bids and integrates a `refund` mechanism for the outbid participants. Time-based elements within the auction are managed using the `getBlockTimestamp` function. Moreover, the contract allows the winning bidders to claim their respective NFTs post-auction through the `claim` function.

   In addition to auction features, the `Converter` contract introduces an unlocking mechanism. It contains functions like `approveUnlock` and `unapproveUnlock`, allowing uToken holders to influence the buyout.

4. The `GovernorAlpha` contract, an adaptation of Compound Finance's governance system, is designed to facilitate decentralized governance through proposing, voting on, and executing community proposals in the Unicly ecosystem. The attributes and functions of the contract are presented in table B.17 and C.1. The contract features mechanisms to ensure fair and transparent governance. This includes a predefined quorum for votes, a threshold for proposal submissions, and a clear timeline for proposal consideration, voting, and execution, managed through the integrated Timelock contract. Functions like *propose*, *queue*, and *execute* enable the community to actively participate in the governance process, from the inception of ideas to their final implementation. The *castVote* and *castVoteBySig* functions further democratize the voting process, allowing for both direct and delegated voting.

5. The `UnicFarm` contract integrates core DeFi functionalities, providing a platform for staking LP tokens to earn UNIC rewards. It is built on a dynamic reward system, where the amount of distributed UNIC tokens varies based on each pool's block number and allocation points. `UnicFarm` attributes and functions can be found in table B.7 and B.8.

   Each pool within `UnicFarm` is defined by its LP token, allocation points, and other pertinent details encapsulated within the `PoolInfo` structure. Users interacting with these pools have their data managed through the `UserInfo` structure, detailing their staked amount and calculated reward debt. The `updatePool` function re-calibrates each pool's state to reflect the latest block number and distribution criteria.

6. The `UnicGallery` contract is designed for users to stake UNIC tokens and earn rewards over time. Table B.9 and B.10 present the smart contract attributes and functions. The core functions, *enter* and *leave*, manage the staking and withdrawal process. Users who enter the gallery lock in UNIC tokens and receive xUnic shares in return. The number of shares received is calculated based on the current ratio of total staked UNIC to xUnic shares. Upon leaving, users redeem their xUnic shares to withdraw a corresponding amount of staked UNIC tokens. The amount withdrawn reflects the initial stake plus any additional rewards accrued during the staking period. `UnicGallery` rewards are implicitly calculated based on the changing ratio of UNIC to xUnic, influenced by the total number of tokens staked in the contract and the overall supply of xUnic shares. This mechanism encourages longer staking periods for potentially higher rewards.

7. The `UnicPumper` enhances the value proposition for xUNIC holders. It converts tokens accrued as fees into UNIC and distributes them as rewards. Table B.11 and B.12 present the attributes and functions of `UnicPumper`. The *convert* function is central to this process, allowing the trade of token pairs for UNIC through a series of swaps, first to WETH and then to UNIC. It follows a methodology similar to SushiSwap's SushiMaker [53].

8. The `UnicSwapV2Factory` contract is instrumental in the Unicly decentralized exchange ecosystem. It empowers users to create liquidity pools for new token pairs, facilitating trading and liquidity provision on the platform. Table B.13 and B.14 presents the attributes and functions of `UnicSwapV2Factory`. The contract's *createPair* function ensures that a corresponding liquidity pool accompanies each new pair.

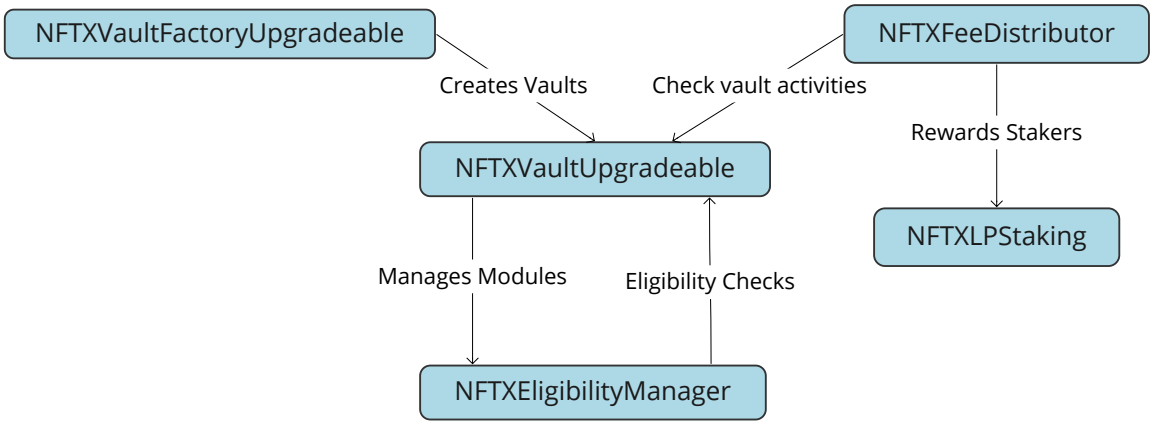


Figure 3.4: NFTX platform smart contract

9. The `Timelock` contract, adapted from Compound Finance's GovernorAlpha [54], is designed to impose a mandatory delay on critical actions. This delay allows stakeholders to review and potentially contest actions before execution. The attributes and functions of `Timelock` contract are presented in table B.15 and B.16. The contract's delay mechanisms are defined by *MINIMUM_DELAY*, *MAXIMUM_DELAY*, and *GRACE_PERIOD*.

3.2.3 NFTX

a. Overview:

Introduced in December 2020, NFTX is a platform designed to create fluid markets for otherwise illiquid Non-Fungible Tokens [55]. Users can deposit their NFTs into an NFTX vault, and in return, they generate a fungible ERC20 token, known as a vToken. This vToken signifies a claim on an arbitrary asset within the vault. Furthermore, vTokens can be employed to reclaim a particular NFT from a vault.

Figure 3.4 illustrates the interaction among the primary smart contracts.

1. Vault Creation and Configuration:

- **Vault Creation:** Users can create vaults for any NFT collection by specifying the NFT asset

address, which points to the smart contract of the NFTs. This process creates a new ERC-20 token (vToken) for that vault.

- **Eligibility Criteria:** Vault creators can set eligibility criteria to specify which NFTs (identified by their token IDs) can be minted into the vault.
- **Vault Features and Fees:** Creators can enable features like minting, random redeems, and targeted redeems. They also set fees for these actions, which are distributed to liquidity providers (LPs) staking their LP tokens in the vault.

2. Minting Process:

- **Minting vTokens:** NFT owners can mint vTokens by depositing their NFTs into a corresponding vault. Each NFT minted into the vault increases the supply of vTokens by one, minus any mint fee set by the vault creator.
- **Mint Fees:** A fee can be charged for minting vTokens, incentivizing initial liquidity provision. This fee is adjustable and can be set to zero during the initial seeding phase of the vault.

3. Redeeming NFTs:

- **Random Redeems:** vToken holders can redeem a random NFT from the vault by spending the corresponding vTokens value.
- **Targeted Redeems:** For an additional fee of 5%, users can select a specific NFT they wish to redeem from the vault. This allows users to acquire desired NFTs directly from the vault.

4. Staking and Rewards:

- **Liquidity Provision:** Users can provide liquidity by adding an equal value of vTokens and a base currency (e.g., ETH) to a liquidity pool on an Automated Market Maker (AMM) like SushiSwap. In return, they receive LP tokens representing their share of the pool.
- **Staking LP Tokens:** LP tokens can be staked in the NFTX platform to earn a portion of the fees generated by the vault activities. Stakers receive xTokens (e.g., xPUNK) representing their staked position.
- **Fee Distribution:** Fees collected from minting and targeted redeems are distributed to stakers.

b. Smart contracts analysis:

1. The *NFTXVaultFactoryUpgradeable* contract utilizes a mapping to manage vaults related to specific assets. Table C.4 and C.5 describes the attributes and functions of the contract. The *createVault* function manages the vault creation process, necessitating essential parameters like the asset's name, symbol, and address. This function is safeguarded by owner permissions and pause conditions, ensuring controlled execution.

   The contract maintains individual and factory-level fee structures for minting, redeeming, and swapping within vaults. The *setFactoryFees* and *setVaultFees* functions, guarded by owner-only access, allow for dynamic adjustment of these fees. Moreover, the contract provides functionalities to manage the ecosystem's broader operational aspects, such as fee distribution and eligibility management. It integrates with `INFTXFeeDistributor` for handling fee distributions and an `eligibility user` to regulate vault participation criteria.

2. The `NFTXVaultUpgradeable` enables users to create fractions for their NFTs. It supports both *ERC721* and *ERC1155* NFT standards, which is indicated by the `is1155` attribute. The key attributes and function of the smart contract are presented in table C.2 and C.3

   One key feature of this contract is handling both single and multiple NFTs through the *mint* function. This is crucial for accommodating various types of NFT collections. The contract also implements a mechanism for redeeming fractions back into NFTs. Moreover, the *swap* function introduces a feature where users can exchange one NFT for another within the same vault. Another aspect is the *setVaultMetadata* function, which is restricted to privileged users and allows for dynamic updates to the vault's metadata.

3. The `NFTXEligibilityManager` inherits from `OwnableUpgradeable`, a governance model where specific actions are restricted to the contract owner. It defines a *EligibilityModule* struct, which includes the implementation address, the target asset address, and a descriptive name. The attributes and functions of the smart contract are described in table C.6 and C.7.

   The *addModule* function allows the owner to introduce new eligibility criteria to the system. It ensures the implementation address is valid and retrieves necessary information from the `INFTXEligibility` interface. The contract also features a method for deploying eligibility clones

using the `ClonesUpgradeable` library. This approach allows for creating eligibility rule instances without deploying a new contract for each instance.

4. The `NFTXLPStaking` contract provides the infrastructure for LP staking. It integrates closely with the `INFTXVaultFactory` to establish staking pools for each vault. Table C.8 and C.9 present the attributes and functions of the smart contract. Key functionalities include the creation and management of staking pools (`addPoolForVault`), reward distribution mechanisms (`receiveRewards`), and user interactions for staking and claiming rewards (`deposit`, `exit`, `claimRewards`). The `onlyAdmin` and `onlyOwner` modifiers ensure robust access control.

5. The `NFTXSimpleFeeDistributor` primary role is to distribute fees accrued in various vaults to designated receivers, including liquidity providers and the treasury, fairly and equitably. Table C.10 and C.11 present the attributes and functions of the smart contract. The contract includes functions such as *distribute*, *addReceiver*, and *removeReceiver* to control the fee flows and ensure that all participants in the ecosystem are rewarded according to their contribution.

3.2.4 Comparison

The comparison between Tessera, Unic.ly, and NFTX platforms shows they have different approaches to NFT fractionalization, catering to market demands and user preferences. Table 3.2 displays the critical aspects of each platform that are compared across multiple dimensions, such as fractionalization technique, governance structure, security features, DeFi integration, and ecosystem impact.

3.3 Standardization proposal

This section defines a standardization proposal for the fractionalization of NFTs. The proposal revolves around defining a set of smart contracts, each with specific roles and interactions, to ensure a secure, standardized, and interoperable framework across different platforms and applications that serve as a baseline for future fractionalization platforms. Figure 3.5 displays the class diagram for our proposed solution, which is further elaborated in the subsequent sections.

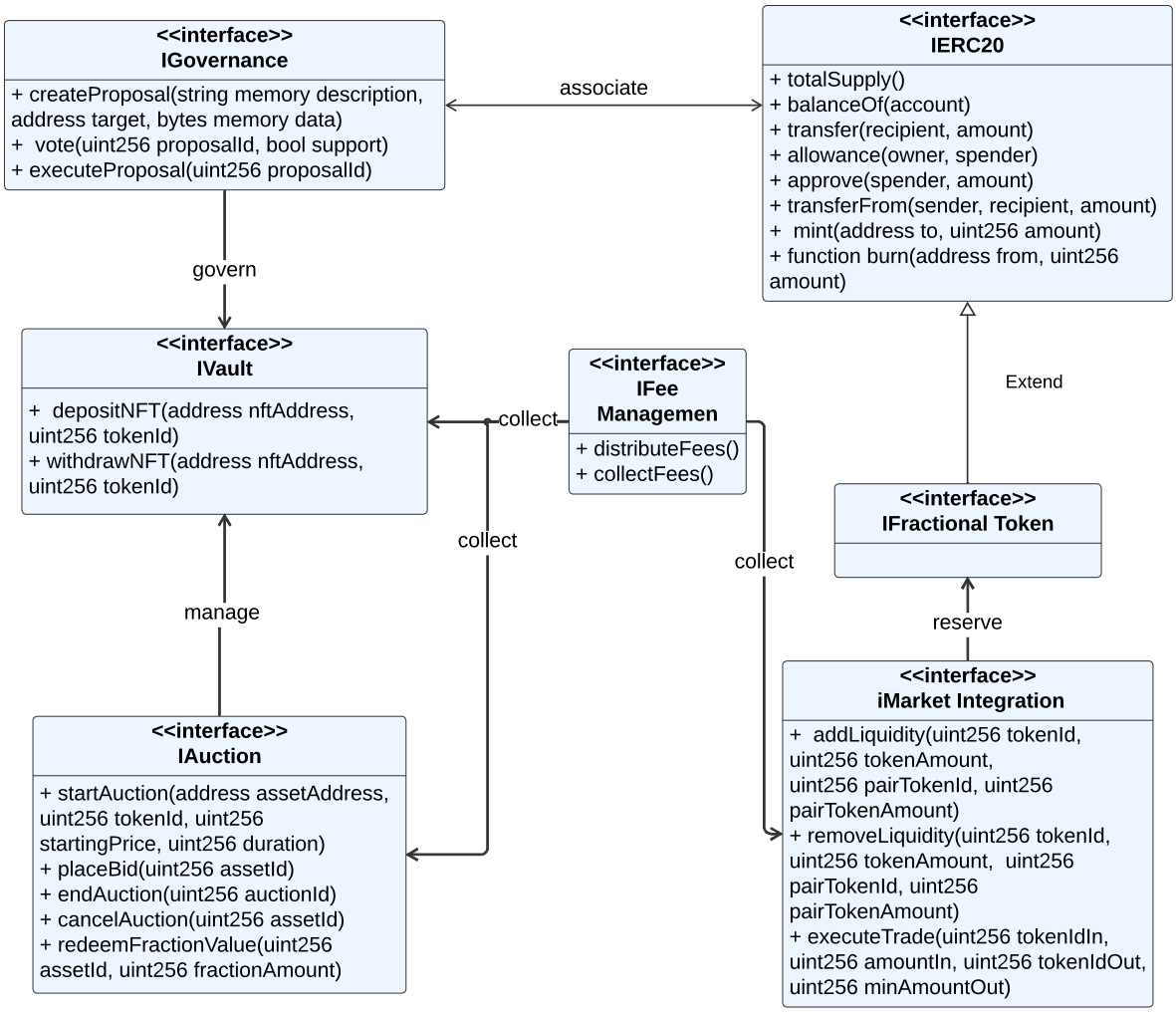


Figure 3.5: Standardization proposal Interfaces

	Tessera	Unicly	NFTX
Supported blockchain	Ethereum	Ethereum	Ethereum
NFT standard	ERC-721/ ERC-1155	ERC-721/ERC-1155	ERC721
Collection support	Yes	Yes	Yes
Methods	Indexation	Aggregation	Indexation
Vote on reserve price	Yes	Yes	No
Buyout start	Initiated by meeting reserve price in ETH	Initiated by bid exceeding current bid and meeting vote threshold	Instant buyout
Buyout Mechanism	Voting Auction	uToken holders vote to unlock the pool	Instant buyout
User documentation	Good	Good	Good
Developer documentation	Medium	Low	Low
Audit	Code Arena Audit [56]	Quantstamp [57]	SECBIT Audit [58] - Trail of Bits Audit [59]
Fee	A dynamic fee from vaulters after an auction is finished.	A fee of 5% from fraction owners when buyout. and 2% auction	5% Minting Fee 5%, 0% Random Redemption Fee, 5% Targeted Redemption Fee (modifiable)
Own token	No	UNIC - governance token capped at 1M total supply	NFTX - governance token capped at 650K total supply
Auction	Yes	No	No
Defi support	No	Yes	Yes

Table 3.2: Comparison between different fractionalization platforms

After analyzing the existing solutions, it becomes evident that most of them share similar roles in the fractionalization process. This typically involves several key smart contracts, including the

Vault Contract, Fractional Token Contract, Auction Contract, Governance Contract, Automated Market Maker (AMM) Contract, and Fee Management Contract.

3.3.1 Vault Contract Interface

The Vault Contract Interface defines the essential functions for securely managing the lifecycle of NFTs within the fractionalization process. It outlines standardized methods for depositing NFTs, minting fractional tokens, and withdrawing NFTs, ensuring interoperability and consistent behavior across different implementations.

a. NFT Deposit Function:

This function transfers an NFT from the depositor's address to the vault. It must verify the caller's ownership of the NFT and initiate the minting of fractional tokens corresponding to the NFT's value.

- **nftAddress:** This parameter specifies the address of the NFT contract. Identifying the specific ERC-721 or ERC-1155 contract that manages the NFT intended for deposit. By providing this address, the Vault Contract can interact directly with the NFT contract to transfer ownership of the NFT into the vault.
- **tokenId:** The unique identifier for the NFT within its contract. The tokenId, with the nftAddress, uniquely identifies the NFT across the Ethereum blockchain.

```
function depositNFT(address nftAddress, uint256 tokenId) external;
```

b. NFT Withdrawal Function:

This function facilitates the withdrawal of an NFT from the vault, enabling fractional token holders to claim the original NFT. The contract must define specific criteria for withdrawal, such as owning a majority of the fractional tokens or auction mechanism.

- **nftAddress:** It is essential to specify the NFT contract from which an NFT will be withdrawn,

just like in the deposit function. This ensures that the correct NFT contract is targeted during the withdrawal operation.

- **tokenId:** This parameter specifies the NFT to be withdrawn based on its unique identifier. It ensures correct transfer based on fractional token ownership.

```
function withdrawNFT(address nftAddress, uint256 tokenId) external;
```

3.3.2 Fractional Token Contract Interface

The Fractional Token Contract Interface represents the shared ownership in an NFT and extends the ERC20 interface. It defines standard operations such as minting new tokens upon NFT deposit, transferring tokens between owners, and burning tokens to adjust ownership shares or upon withdrawal of the NFT from the vault. Upon a successful NFT deposit, this function is invoked to mint the corresponding amount of fractional tokens, which are then allocated to the ‘to‘ address.

a. Minting Fractional Tokens:

Minting creates new fractional tokens, distributed to an NFT's depositor, representing their ownership share.

- **to:** The address that will receive the minted tokens, typically the depositor's address. This parameter ensures the correct allocation of ownership shares in the form of fractional tokens.
- **amount:** Specifies the number of fractional tokens to mint, which correlates to the value of the deposited NFT.

```
function mint(address to, uint256 amount) external;
```

b. Transferring Fractional Tokens:

The ability to transfer fractional tokens between addresses is fundamental for trading and redistributing ownership shares. This function facilitates the secure and flexible redistribution

of fractional ownership, allowing token holders to engage in transactions that reflect changes in ownership interests.

- **from:** The current holder's address of the fractional tokens. Ensuring that tokens are correctly debited from the rightful owner is crucial.
- **to:** The recipient's address of the tokens, enabling the transfer of ownership shares.
- **amount:** The number of tokens to be transferred, dictating the size of the ownership share being moved.

```
function transfer(address from, address to, uint256 amount) external;
```

c. Burning Fractional Tokens:

Burning fractional tokens is a mechanism to reduce the total supply, typically used when an NFT is withdrawn. The burning process ensures the fractional token supply remains accurate and reflects the NFT's current fractional ownership distribution.

- **from:** Indicates the address from which tokens will be burned, aligning the token supply with the current ownership structure.
- **amount:** The number of tokens to be burned directly impacts the representation of ownership shares within the ecosystem.

```
function burn(address from, uint256 amount) external;
```

3.3.3 Auction Contract Interface

The Auction Contract facilitates competitive bidding processes for fractional tokens or entire NFTs. This contract manages auctions, tracks bids, determines winners, and ensures the transfer of ownership following auction conclusions.

a. Initiating an Auction:

To start an auction, the NFT owner or a fractional token holder specifies the asset to be auctioned, the starting price, and the auction duration.

```
function startAuction(address assetAddress, uint256 tokenId, uint256
startingPrice, uint256 duration) external;
```

- **assetAddress:** The contract address of the asset being auctioned, which could be the Vault Contract for an NFT.

- **assetId:** For an NFT, this is the tokenId; - **startingPrice:** The minimum bid required to participate in the auction. - **duration:** The time frame for placing bids.

b. Placing Bids:

Participants can place bids on an active auction by sending the bid amount in the transaction, which must be higher than the current highest bid.

```
function placeBid(uint256 assetId) external payable;
```

- **assetId:** The assetId of the item in the auction - The sent value (*msg.value*) must exceed the current highest bid for the bid to be considered valid.

c. Concluding the Auction:

After the auction duration has elapsed, the contract concludes the auction, transferring the auctioned asset to the highest bidder and redistributing the bid amounts accordingly. This function ensures that the highest bidder receives the asset, returning the bids to unsuccessful bidders and transferring the winning bid amount to the asset's seller.

```
function endAuction(uint256 auctionId) external;
```

d. Canceling an Auction:

The Auction Contract also provides functionality to cancel an ongoing auction. This action can be initiated by the contract's governance mechanism under specific circumstances, such as detecting fraudulent activity or at the asset owner's request for legitimate reasons. Upon cancellation, the auction state is updated to reflect that it is no longer active, and any bids placed during the auction

are refunded to the respective bidders.

```
function cancelAuction(uint256 assetId);
```

assetId: The ID of the NFT whose auction is to be canceled. This function SHOULD only be called by the governance contract or an account with equivalent permissions, ensuring that the cancellation process is controlled and secure.

e. Redeeming Fractional Value:

After the sale of an NFT, fractional token holders are entitled to redeem their share of the sale proceeds. The redeem function calculates the fractional tokens' value based on the NFT's total sale proceeds and transfers the corresponding Ether amount to the token holder. Fractional tokens SHOULD be burned upon redemption to prevent double-spending.

```
function redeemFractionValue(uint256 assetId, uint256 fractionAmount) external;
```

assetId: The ID of the asset whose sale proceeds are being claimed. **fractionAmount:** The amount of fractional tokens the holder wishes to redeem.

3.3.4 Market Integration Contract Interface

The Market Integration Contract Interface streamlines decentralized finance (DeFi) ecosystem interactions for a broad spectrum of ERC20 tokens. This interface enables users to effectively manage liquidity pools, execute trades, and interface with various DeFi platforms, thereby enhancing the utility and liquidity of ERC20 tokens.

a. Liquidity Management:

This functionality permits users to add to or withdraw from the liquidity of any ERC20 token pair.

```
function addLiquidity(uint256 tokenId, uint256 tokenAmount,
uint256 pairTokenId, uint256 pairTokenAmount) external;
```

```
function removeLiquidity(uint256 tokenId, uint256 tokenAmount,
uint256 pairTokenId, uint256 pairTokenAmount) external;
```

- **tokenId, pairTokenId:** Identifiers for the ERC20 tokens involved in the liquidity transaction.
- **tokenAmount, pairTokenAmount:** Quantities of the respective ERC20 tokens to be added to or removed from the liquidity pool.

b. Trade Execution:

Enables swapping one ERC20 token for another within the pool to leverage market dynamics and diversify their asset holdings efficiently.

```
function executeTrade(uint256 tokenIdIn, uint256 amountIn, uint256 tokenIdOut,
uint256 minAmountOut) external;
```

- **tokenIdIn:** The identifier of the ERC20 token being offered in the exchange.
- **amountIn:** The volume of the input ERC20 token being traded.
- **tokenIdOut:** The identifier of the ERC20 token desired from the trade.
- **minAmountOut:** The minimum acceptable amount of the output ERC20 token ensures the trade is executed within acceptable slippage limits.

c. Interfacing with External Platforms:

The interface supports interactions with various DeFi platforms, enabling users to leverage their ERC20 tokens across multiple protocols for liquidity provision, trading, and other financial activities.

c. Security and Compliance:

Adhering to high standards of security and compliance is paramount. The contract employs mechanisms to ensure the safety of users' assets and the integrity of transactions, facilitating a trustworthy environment for engaging with the DeFi landscape.

This Market Integration Contract Interface extends the functionality and applicability of ERC20 tokens within the broader DeFi ecosystem. It provides a robust framework for liquidity management and trade execution, enabling token holders to maximize the potential of their digital assets securely and competently.

3.3.5 Governance Contract Interface

The Governance Contract for Fraction Holders is specifically designed to facilitate a democratic governance process. It allows fraction holders to propose, vote on, and implement changes. This system ensures that decision-making is aligned with the community's interests, leveraging ERC20 tokens for voting power.

a. Proposal Creation and Management:

Fraction holders can initiate governance proposals subject to a community vote. Proposals may include changes to protocol parameters, upgrades, or other significant decisions.

```
function createProposal(string memory description,
address target, bytes memory data) public;
```

- **Description:** A brief overview of the proposal's purpose and objectives. - **Target:** The contract address that the proposal aims to interact with. - **Data:** The encoded function calls data required for the proposal's execution.

b. Voting Mechanism:

The contract allows token holders to cast votes on active proposals. The voting power of each holder SHOULD be proportional to their token ownership, ensuring a fair and representative governance process.

```
function vote(uint256 proposalId, bool support) public;
```

- **ProposalId:** The unique identifier of the proposal being voted on. - **Support:** A boolean value indicating whether the vote is in favor of (true) or against (false) the proposal.

c. Proposal Execution:

After the conclusion of the voting period, proposals that meet the required quorum and approval thresholds are executed, enacting the proposed changes within the system. Execution is contingent upon meeting predefined conditions, such as quorum and majority support.

```
function executeProposal(uint256 proposalId) public;
```

3.3.6 Fee Management Contract Interface

The Fee Management Contract Interface is designed to handle the financial aspects of the NFT fractionalization platform, explicitly focusing on the collection and distribution of fees. This contract ensures that fees generated from various operations within the ecosystem are managed transparently and efficiently.

a. Collecting Fees:

Fees are collected from various activities, such as NFT transactions, fractional token trades, and other services the platform provides. This function is responsible for accumulating user fees and securely storing them within the contract for future distribution or reinvestment into the platform.

```
function collectFees() external payable;
```

b. Distributing Fees:

The collected fees are distributed according to predefined rules, including rewarding token holders, covering operational costs, or funding community proposals. This function outlines the mechanism for allocating the collected fees, ensuring stakeholders are compensated for their participation and investment in the platform.

```
function distributeFees() external;
```

3.3.7 Security and Functional Properties

a. Essential Properties:

To ensure the platform's compatibility, security, and comprehensive functionality, the fractionalization abstraction adheres to the following properties, inspired by the principles underlying the ERC-721 [17] and ERC-20 [16] standards:

1. **Distinct Standard**: Facilitate easy identification of the fractional standard by third parties, distinguishing it from other token standards for seamless integration and interaction.
2. **Asset Traceability**: Enable access to the associated NFT's address and tokenID for third parties, ensuring transparency and ease of integration with external platforms and services.
3. **Proportional Gains Distribution**: Ensure proportional distribution of any revenue generated from the NFT based on the ownership share in the fractional tokens, promoting fairness and transparency in profit sharing.
4. **Prevent Double Withdrawal**: Implement safeguards to prevent users from claiming their share of the proceeds more than once, protecting against double withdrawal vulnerabilities.
5. **Governance Participation**: Allow fractional token holders to participate in governance decisions regarding the NFT, including its sale or use, reinforcing the decentralized nature of asset management.
6. **Fraud and Tamper Resistance**: Incorporate security measures to protect against unauthorized access, fraud, and tampering, maintaining the integrity and trustworthiness of the fractionalization process.

b. Verification of Properties:

Developers aiming to respect the outlined properties in Section 3.3.7 should diligently implement the following verification, while users are encouraged to confirm these aspects before engaging with the solution:

1. **Asset Traceability**: Stores the NFT address and ID within the smart contract, providing getter functions for external access, thus ensuring transparency regarding the underlying asset.
2. **Proportional Gains Distribution**: Leverages the ERC-20 *totalSupply* and *balanceOf* functions to proportionally distribute gains among fractional token holders based on their ownership

stake.

3. **Prevent Double Withdrawal**: Ensure the smart contract executes the *burn* function before sending gains to fractional holders during the redemption process. Burning tokens permanently removes them from circulation, eliminating the risk of withdrawing gains more than once.

4. **Governance Participation**: Implements a voting system where fractional holders can vote on NFT management proposals, ensuring democratic decision-making processes. And create a modifier for setter functions that require governance.

5. **Fraud and Tamper Resistance**: Incorporates security practices such as access controls, audit trails, and smart contract audits to safeguard against unauthorized actions and ensure the contract's integrity.

These properties and their verification methods establish a secure and functional framework for NFT fractionalization, aiming to maximize compatibility with existing standards and protocols while providing a comprehensive set of asset management and trading features.

3.4 Standard evaluation

In our evaluation of the proposed standard for NFT fractionalization, we noted the diverse approaches across existing platforms, each characterized by unique implementation details, features, and security protocols. While these innovations contribute positively, they also lead to a fragmented ecosystem. This fragmentation led developers to start from scratch with each new solution, significantly increasing both time and financial investments.

Our primary objective was to extract the essential components of the fractionalization process into a unified standard. We conducted a detailed manual inspection of smart contracts from leading platforms such as Tessera, Unicly, and NFTX. Our analysis focused on comparing their functionalities to extract the core elements of the fractionalization process, as detailed in Section 3.2. This thorough investigation forms the basis of our proposal, ensuring that the standard is deeply rooted in proven fractionalization processes.

To further assess our standard, we designed various scenarios that could be presented using our standard and proposed a concrete implementation. These scenarios are directly inspired by the three studied solutions, demonstrating that our standard can effectively reproduce the functionalities found in existing solutions. This alignment confirms the adaptability and applicability of our standard across different platform requirements. The detailed descriptions of these scenarios and their implementations are presented in Chapter 5.

This comprehensive evaluation has demonstrated the viability of our proposed standard, confirming its potential to significantly enhance the NFT fractionalization landscape by providing a framework that is more secure, efficient, and user-oriented.

3.5 Conclusion

The proposed standardization for NFT fractionalization aims to create a robust, secure, and interoperable framework. By delineating clear roles, interactions, and technical specifications for the involved smart contracts, this proposal seeks to enhance the functionality, accessibility, and liquidity of the NFT market, paving the way for a more inclusive and dynamic digital asset ecosystem.

Chapter 4:

Vulnerabilities of smart contracts and mitigation schemes

In this chapter, we offer an analysis that combines a literature review with experimental insights aimed at assisting developers in implementing secure smart contracts. We provide a list of common vulnerabilities and corresponding mitigation strategies and examine their impact on the NFT and fractionalization market. Additionally, we assess the efficacy of widely adopted community tools through execution and testing on sample smart contracts.

4.1 Survey Methodology

In this section, we present the primary research questions that we hope to answer through this survey. The methodology we used to collect existing related work is then presented.

This survey aims to answer the following research questions:

- RQ1: What are the most frequent vulnerabilities in Solidity smart contracts?
- RQ2: How to mitigate vulnerabilities in Solidity smart contracts?
- RQ3: What are the existing methodologies used to detect the vulnerabilities of smart contracts, and how do they compare?
- RQ4: what are the most used tools by the Ethereum community to investigate/mitigate smart contracts based on Github forks and web articles?

To address these questions, we conducted a Multivocal Literature Review (MLR) [60] for data

preparation; This technique incorporates both gray literature and white literature. Gray literature includes blogs, videos, and forums; practitioners from industry and academia usually write it, and it's not peer-reviewed. White literature contains peer-reviewed research articles from journals. We chose the published studies in journals and conferences with high-impact factors and competitive acceptance rates. We also check the citation count of the studies being chosen on Google Scholar to evaluate their impact on the evolution of this emerging paradigm. The rationale behind choosing MLR is that the field of smart contracts security is still relatively new; thus, including literature from practitioners is helpful in providing a complete overview. Moreover, the feedback provided by smart contract developers is crucial in selecting the most valuable tools.

As for the search strategy, we followed a protocol presented by Kitchenham for systemic reviews [61]. This protocol consists of three phases:

(a) Define a search string

(b) Use the string in search engines

(c) Select literature based on predefined inclusion and exclusion criteria.

We defined a search string, as presented in Listing 1, to gather resources on the following topics:

(a) Review articles about smart contract security

(b) Papers discussing vulnerabilities in Ethereum smart contracts

(c) Papers covering detection methodologies or mitigation schemes of smart contract vulnerabilities

(d) Papers about tools that detect vulnerabilities in smart contracts.

We have only included papers containing the keywords in the subject, title, or abstract. "business" is used because smart contracts often handle business logic. The term "opcode" refers to the basic instructions executed by the blockchain's virtual machine, which regulates the logic and operations of the smart contract. "Turing completeness," on the other hand, refers to the capacity of a blockchain or smart contract language to simulate any Turing machine, enabling it to resolve any computational problem given sufficient time and resources. These terms are employed to represent

smart contracts.

```
(( List* OR detect* OR spot* OR extract* OR find* OR identif*)  AND ( vulnerabilt* OR bug* OR attack* OR
↪  issue* OR securit*)  AND ("ethereum")  AND(smart contract* OR business* opcode*OR Turing complet*) )
```

Listing 1: Used search string

Figure 4.1 illustrates the complete process of article identifications. 504 conference and journal articles were found using the specified search string. 41 articles are included based on the exclusion and inclusion criteria mentioned in table 4.1. As for gray literature, we focused on blog posts found within the first eight pages of Google search results. The search strings included "Ethereum smart contract vulnerabilities and mitigation" and "Ethereum smart contract analysis tools." Out of the numerous blogs reviewed, eight posts met our established criteria. To assist the quality of the gray literature, we have taken into consideration the following aspects:

- **The publisher's reputation :** Evaluate the publisher's standing in the industry by checking for recognition within blockchain and cybersecurity communities, as well as the level of interaction with their blogs or reports.

- **The author's expertise in smart contract security**: Verify the author's credentials by examining their professional background, primarily if their job position is related to blockchain and smart contract security. Look for evidence of their expertise, such as previous publications, participation in industry forums, or contributions to blockchain projects.

- **The content is clearly stated with supported arguments:** Assess the clarity of the content by checking if the arguments are logically structured and if the article avoids ambiguous terminology without adequate explanation.

In the next sections , we present some of the common vulnerabilities of Ethereum smart contracts alongside real-world attacks and prevention mechanisms.

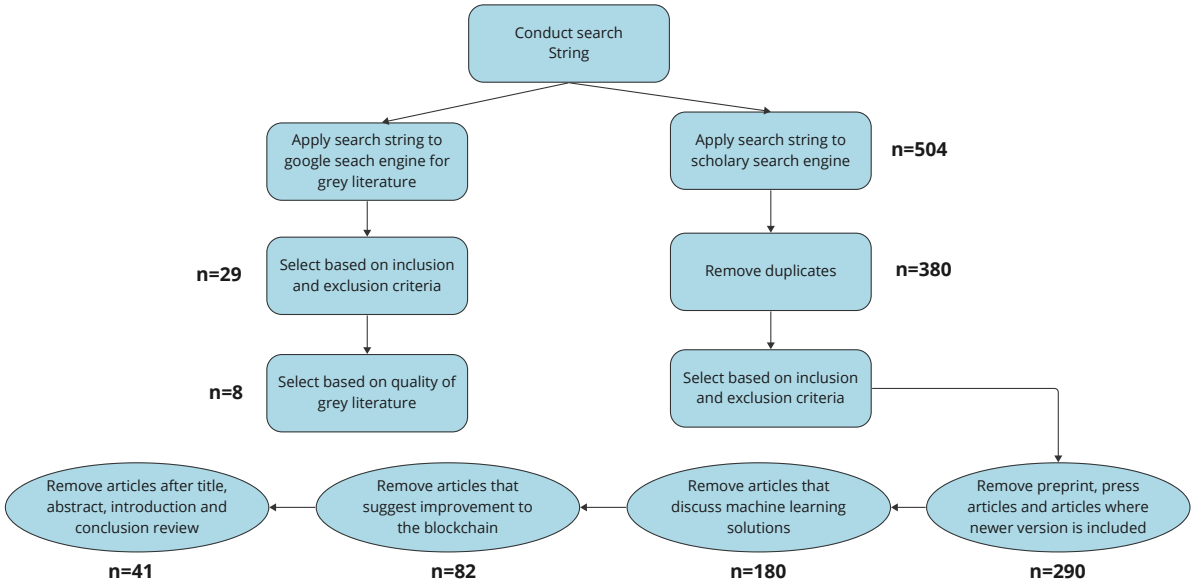


Figure 4.1: Articles identification, exclusion and inclusion methodology.

4.2 Reentrancy

4.2.1 Description

Reentrancy is one of the most critical vulnerabilities to address when implementing smart contracts, also known as a recursive call attack. A contract calling another contract, with external calls, will cause the stoppage of the calling contract's execution and memory state until the called function returns a response. External calls are exposed in contract interfaces; hackers can use them to invoke a function within the contract numerous times, causing the contract to perform unanticipated tasks. This vulnerability occurs in a solidity smart contract performing critical tasks (e.g., token transfer) before resolving the effects that should have been addressed (e.g., balance update).

4.2.2 Implications on NFT Fractionalization

Reentrancy vulnerabilities have already significantly impacted decentralized finance (DeFi) protocols, illustrating potential risks for fractionalized NFT platforms. For instance, the dForce DeFi Protocol Hack [62] in February 2023, where an attacker exploited a reentrancy vulnerability in the

Criteria	Description
Inclusion Criteria	- Studies related to vulnerabilities in Ethereum: Vulnerabilities explication and mitigation techniques. - Studies related to the security improvement of Ethereum smart contracts. - Studies related to vulnerability detection tools in Ethereum smart contracts. - Studies that introduce open-source tools to detect the vulnerabilities of solidity smart contracts.
Exclusion Criteria	- Non-English papers. - Results past the first eight Google pages, as we have noticed that after the eight pages, the results are not relevant. - Data sets, tweets, presentations. - Tools that use machine learning, such as comparing machine learning tools, need different criteria, such as the used training models, accuracy, etc. - Studies that are based on the improvement of blockchain infrastructure rather than smart contract programming - A study that is an older version of another paper that has previously been considered.

Table 4.1: Exclusion/inclusion criteria

Curve Finance vault on the Arbitrum and Optimism blockchains, part of the dForce protocol, led to the theft of approximately $3.6 million in assets. This attack underscores the danger reentrancy poses to smart contracts handling complex financial transactions, serving as a warning for platforms dealing with fractionalized NFTs. Furthermore, the CREAM Finance Hack [63] in August 2021 and the Siren Protocol Hack [64] in September 2021, with losses of $18.8 million and $3.5 million, respectively, further exemplify the ongoing threat of reentrancy attacks to the blockchain and DeFi sectors.

Reentrancy attacks can distort market dynamics by unfairly redistributing assets, leading to market manipulation and loss of liquidity. For fractionalized NFTs, where valuation depends on the underlying asset's perceived value and the platform's integrity, such attacks can result in volatile price swings and diminished market confidence.

4.2.3 Protection Measures

To mitigate reentrancy attacks, it is advisable to use the *send()* and *transfer()* methods for transferring funds instead of the more general *call()* method. These methods are considered safer because they impose a gas limit of 2,300 gas, effectively restricting the called contract from performing complex operations, including additional external calls that could lead to reentrancy.

The *checks-effects-interactions* pattern [65] represents the most reliable approach to prevent reentrancy attacks. This design pattern organizes the code of a function to minimize unwanted side effects by ensuring that:

1. All conditions and validations (*require()* checks) are performed upfront.
2. State changes (e.g., balance updates) are applied.
3. Interactions with other contracts occur last, after all checks and state changes, to prevent any possibility of reentrancy affecting the contract's state.

Employing a mutex (mutual exclusion) is a robust mechanism against reentrancy. A mutex locks the contract's state during the execution of critical code sections, allowing only one operation at a time. This ensures that if an attacker attempts a reentrancy attack, the contract state remains locked from the previous call, preventing unauthorized state changes. Care must be taken to release the mutex after use to avoid deadlocks. OpenZeppelin's *ReentrancyGuard* [66] provides an implementation of this concept through the *nonReentrant* modifier, adding layer of security for functions vulnerable to reentrancy.

4.2.4 Example

Listing 2 illustrates a simplified smart contract, FractionalNFT, designed to manage fractional shares of an NFT's revenue. This contract allows owners of fractional shares to withdraw their proportion of sales revenue when the NFT is sold. The contract contains a critical flaw in its *withdrawShare* function, which makes it susceptible to reentrancy attacks. In this contract, the *withdrawShare* function fails to adhere to the checks-effects-interactions pattern, updating the

shareholder's balance *after* transferring funds. This ordering allows for the potential reentrancy, where a malicious actor could recursively call *withdrawShare* within a fallback function, draining the contract's funds beyond their rightful share.

```
pragma solidity ^0.8.0;
contract FractionalNFT {
    mapping(address => uint256) public shareBalances;
    uint256 public totalRevenue;


    function withdrawShare() public {
        uint256 share = shareBalances[msg.sender];
        require(share > 0, "No shares owned.");
        require(totalRevenue >= share, "Insufficient revenue.");

        (bool sent, ) = msg.sender.call{value: share}("");
        require(sent, "Ether transfer failed.");

        shareBalances[msg.sender] = 0;
        totalRevenue -= share;
    }
}
```

Listing 2: Simplified contract with reentrancy vulnerability

To address this vulnerability, the *ReentrancyGuard* utility from the OpenZeppelin security library can be integrated to prevent recursive calls, as demonstrated in Listing 3. By employing the *nonReentrant* modifier provided by *ReentrancyGuard*, the revised contract ensures that the *withdrawShare* function cannot be re-entered while it is still executing, effectively mitigating the reentrancy vulnerability.

4.3 Front Running

4.3.1 Description

Front-running, also known as transaction ordering dependency, occurs when the execution logic depends on the order of the submitted transactions. The miner [67] determines the order of transactions in Ethereum. The transaction is visible to the network before being executed. The

```
pragma solidity ^0.8.0
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
contract SafeFractionalNFT is ReentrancyGuard {
    mapping(address => uint256) public shareBalances;
    uint256 public totalRevenue;

    function withdrawShare() public nonReentrant {
        uint256 share = shareBalances[msg.sender];
        require(share > 0, "No shares owned.");
        require(totalRevenue >= share, "Insufficient revenue.");

        (bool sent, ) = msg.sender.call{value: share}("");
        require(sent, "Ether transfer failed.");

        shareBalances[msg.sender] = 0;
        totalRevenue -= share;
    }
}
}

```

Listing 3: Simplified contract that mitigates reentrancy vulnerability

participants exploit this visibility by sending transactions with a higher gas price to be included first.

4.3.2 Implications on NFT Fractionalization

Front-running can have profound implications in the context of NFT fractionalization that might include:

1. **Manipulation of Share Prices:** Malicious actors could exploit transaction ordering to manipulate the market for fractional tokens, buying up tokens at lower prices before a large transaction increases their value or, conversely, selling tokens to depress prices ahead of a significant sell order.

2. **Interference with Auction Mechanisms**: Many NFT platforms use auction mechanisms for selling fractional shares or entire NFTs. Front-runners could preemptively place bids to disrupt fair auction outcomes, affecting the final sale price of an NFT.

3. **Unfair Distribution of Revenue**: In scenarios where revenue from NFT sales is distributed among share owners, front-runners could strategically insert transactions to claim a disproportionate share of the distributions, undermining the fairness of the process.

4.3.3 Protection Measures

The best solution to protect against front-running vulnerability is to remove the advantage of transaction ordering from the application. Another possible solution is to remove the importance of time. Another possible solution is to use a commit-reveal hash scheme where the participant submits the hash of the answer instead of the answer. The contract then stores the hash and the sender's address; the answer is revealed only after all the responses are submitted.

4.3.4 Example

Listing 4 illustrates a smart contract managing the sale of fractional tokens of an NFT, vulnerable to front-running. In this scenario, an attacker could observe pending purchase transactions and execute their purchase with a higher gas fee, securing shares at the current price before a large purchase increases their value, thereby disadvantaging legitimate buyers.

```
1
2   contract FractionalNFTToken {
3       uint public tokenPrice;
4       uint public availableTokens;
5       address owner;
6
7       function buyTokens(uint quantity) public payable {
8           require(msg.value >= quantity * tokenPrice, "Insufficient payment");
9           require(quantity <= availableShares, "Not enough shares available");
10          availableShares -= quantity;
11          // Transfer logic for shares not shown for brevity
12      }
13  }
```

Listing 4: Simplified contract with Front Running vulnerability

The contract can be revised to include a commit-reveal scheme to mitigate the vulnerability and ensure fair transaction processing. This modification requires buyers to commit to a purchase without initially revealing the exact quantity or price, followed by a reveal phase where the transaction details are disclosed. A Mitigation of the front-running attack is shown in Listing 5.

```
1
2   contract SecureFractionalNFTTokens is ReentrancyGuard {
3       uint public tokenPrice;
4       uint public availableTokens;
5       address owner;
6       mapping(bytes32 => bool) public commitments;
7       uint public revealEndTime;
8       bool public saleStarted;
9       function commitToBuy(bytes32 commitment) public {
10          require(block.timestamp < revealEndTime, "Commit phase is over");
11          commitments[commitment] = true;
12      }
13      function revealBuy(uint quantity, uint maxPrice, bytes32 nonce) public payable nonReentrant {
14          require(saleStarted && block.timestamp >= revealEndTime, "Reveal phase not started");
15          bytes32 commitment = keccak256(abi.encodePacked(msg.sender, quantity, maxPrice, nonce));
16          require(commitments[commitment], "Invalid commitment");
17          require(maxPrice >= tokenPrice, "Share price exceeded max price");
18          require(msg.value >= quantity * tokenPrice, "Insufficient payment");
19          require(quantity <= availableTokens, "Not enough shares available");
20          availableTokens -= quantity;
21          // Logic to handle purchase and transfer token
22      }
23  }
```

Listing 5: Simplified contract that mitigates Front Running vulnerability

4.4 Arithmetic

4.4.1 Description

Solidity's integer types, such as `uint8`, `uint16`, and `uint256`, are constrained by fixed sizes, capable of representing numbers within specific ranges. Arithmetic operations that result in values outside these permissible ranges lead to integer overflow (exceeding the maximum value) or underflow (dropping below zero) [68]. These vulnerabilities can cause smart contracts to behave unpredictably, potentially leading to significant security breaches or financial losses.

4.4.2 Implications on NFT Fractionalization

In the context of NFT fractionalization, arithmetic vulnerabilities pose unique challenges. For instance, when distributing sales revenue among shareholders, overflow or underflow errors can lead to incorrect allocation of funds. This affects the fairness and accuracy of fee distribution and exposes the platform to potential exploitation. Arithmetic vulnerabilities, particularly overflows and underflows, have had substantial consequences in the decentralized finance (DeFi) sector,

highlighting the associated risks for platforms dealing with fractionalized NFTs. A prominent example includes the *mintToken* function in the Coinstar (CSTR) Ethereum token's smart contract [69], which suffered from an integer overflow. This flaw permitted the contract owner to adjust any user's balance to any chosen value arbitrarily. Another example is the *4chan gang group* experienced a substantial loss of $2.3 million due to an underflow vulnerability in the ERC-20 token implementation of PoWH (Proof of Weak Hands), which allowed an attacker to exploit this flaw for financial gain [70].

4.4.3 Protection Measures

Arithmetic operations should be appropriately implemented by checking the operators and operands before operating to avoid integer overflows and underflows. It is recommended to use the *assert()* and *require()* modifiers. Using the library for arithmetic functions is also advisable, called *SafeMath* by OpenZeppelin [68]. Solidity version 0.8.0 has included this library, so the transaction will revert if an overflow/underflow occurs. A prevalent method for detecting this type of vulnerability involves taint analysis, a technique we will elaborate on in Section 4.9.

4.4.4 Example

Listing 6 showcases a smart contract managing fractional tokens of an NFT, vulnerable to underflow. If an attempt is made to transfer more tokens than available under Solidity versions before 0.8.0, this could lead to an underflow, setting the token balance to an incorrect high value and potentially enabling malicious token distribution.

```

contract FractionalNFT {
    mapping(address => uint256) public TokensOwned;
    uint256 public totaltokens = 1000; // Total tokens available for the NFT

    function transferTokens(address to, uint256 amount) public {
        // This subtraction could cause an underflow if the sender doesn't have enough tokens
        TokensOwned[msg.sender] -= amount;
        TokensOwned[to] += amount;
    }
}
```

Listing 6: Simplified contract with Arithmetic vulnerability

Given Solidity 0.8.0's automatic checks for arithmetic operations, the compiler directly addresses the primary vulnerability of underflows and overflows. However, we can implement additional logic further to safeguard the integrity of fractional token calculations and transfers. An improved version of the smart contract is presented in Listing 7.

```
contract SecureFractionalNFT {
    mapping(address => uint256) public TokensOwned;
    uint256 public totalShares = 1000; // Total tokens available for the NFT

    function transferTokens(address to, uint256 amount) public {
        require(sharesOwned[msg.sender] >= amount, "Not enough shares");
        require(to != address(0), "Invalid recipient");

        uint256 senderFinalTokens = sharesOwned[msg.sender] - amount;
        uint256 recipientFinalTokens = sharesOwned[to] + amount;
        require(senderFinalTokens + recipientFinalTokens == sharesOwned[msg.sender] + sharesOwned[to],
↪  "Share transfer error");

        sharesOwned[msg.sender] = senderFinalTokens;
        sharesOwned[to] = recipientFinalTokens;
    }
}
```

Listing 7: Simplified contract that mitigates arithmetic vulnerability

4.5 Mishandled Exceptions

4.5.1 Description

Solidity provides two primary paradigms for interacting with external contracts: direct contract calls and low-level calls. Direct contract calls involve invoking functions directly on known contract interfaces, which inherently revert to failure, thereby throwing an exception. Conversely, low-level calls, executed via methods like `call()`, `delegatecall()`, and `callcode()`, return a boolean success flag instead of reverting on exceptions [30]. These calls do not inherently revert transaction execution upon failure; instead, they return `false`, necessitating explicit checks of their return values to ensure the intended execution flow. Failure to adequately check the result of a low-level call can lead to unintended execution continuation, potentially compromising contract logic and security. This vulnerability was notably exploited in the King of Ether game, where the smart contract's failure to verify the result of a `send()` operation led to discrepancies in payments, resulting in users overpaying or underpaying [71].

4.5.2 Implications on NFT Fractionalization

Mishandled exceptions, notably in operations involving the transfer of tokens or the distribution of revenues, pose significant risks to the platform's functionality. For example, if a smart contract designed to distribute sales revenue from an NFT among its token holders neglects to confirm the success of these transactions, it could lead to financial disparities. Such scenarios may arise when the contract employs low-level calls for fund transfers without verifying their execution success. Malicious entities might seize on these vulnerabilities, intentionally causing transactions to fail silently by making a contract reject transactions in its fallback function.

4.5.3 Protection Measures

Invoking functions in smart contracts, either through direct or low-level calls like call or *delegatecall*, requires careful handling of potential exceptions. Employing exception-handling operators such as require, revert, and assert enables conditional checks and allows for the reversion of transactions with clear error messages when conditions fail. Checking the success of low-level calls and setting up safe fallback mechanisms are crucial.

4.5.4 Example

The example 8 shows a contract that manages fractional ownership of an NFT. It includes a function to distribute proceeds from NFT sales to token holders. The contract incorrectly handles a low-level call when sending proceeds, posing a risk if the call fails.

In the revised contract 9, we use a safer approach to distribute proceeds by checking the success of each payment and reverting the transaction if a payment fails.

```
contract FractionalNFT {
    mapping(address => uint256) public ownershipTokens;
    uint256 public totalTokens = 10000;
    address payable[] public shareholders;

    // Function to distribute sales proceeds to shareholders
    function distributeProceeds() public payable {
        for(uint i = 0; i < shareholders.length; i++) {
            address payable shareholder = shareholders[i];
            uint256 amount = msg.value * ownershipTokens[shareholder] / totalTokens;
            // Vulnerable low-level call without checking success
            shareholder.call{value: amount}("");
        }
    }
}
```

Listing 8: Simplified contract with mishandled exceptions

```
contract SecureFractionalNFT {
    mapping(address => uint256) public ownershipTokens;
    uint256 public totalTokens = 10000;
    address payable[] public shareholders;

    // Function to securely distribute sales proceeds to shareholders
    function distributeProceeds() public payable {
        for(uint i = 0; i < shareholders.length; i++) {
            address payable shareholder = shareholders[i];
            uint256 amount = msg.value * ownershipTokens[shareholder] / totalTokens;
            // Securely sending proceeds and checking for success
            (bool success, ) = shareholder.call{value: amount}("");
            require(success, "Failed to send proceeds");
        }
    }
}

```

Listing 9: Simplified contract that mitigates mishandled exceptions

4.6 Code Injection via delegatecall

4.6.1 Description

There is a unique method called a delegate call. The DELEGATECALL opcode is similar to a conventional message call, except that the code performed at the target address is executed in the context of the calling contract. The current address, storage, and balance refer to the calling contract. This enables a smart contract to load code from another smart contract at runtime dynamically. Calling into untrusted contracts via *delegatecall()* is particularly risky since the code at the target smart contract has complete control over the caller's balance; thus, it can modify any

of the caller's storage data.

4.6.2 Implications on NFT Fractionalization

Malicious code executed through delegatecall might tamper with the contract's ownership or control mechanisms, enabling attackers to reroute assets or funds. Furthermore, attackers could change the revenue distribution logic, redirecting profits meant for rightful owners to unauthorized entities. In extreme cases, like the second Parity multi-sig attack where an attacker took over three main Parity wallets and stole $31 million [72], these vulnerabilities could let attackers take over the NFT fractionalization contract entirely. They could push out the real owners and managers, potentially locking, freezing, or stealing assets.

4.6.3 Protection Measures

When employing *delegatecall()*, caution is paramount, especially when dealing with contracts not fully trusted. It's critical to avoid making delegatecall() to addresses derived from user inputs unless there's a rigorous verification process against a list of trusted contracts. This precaution helps ensure that only known, secure contracts can be interacted with, significantly reducing the risk of malicious interference.

Solidity's *library* keyword facilitates the creation of library contracts designed to be stateless and immune to destruction. By defining a contract as a stateless library, it's guaranteed that the executing code cannot alter the storage data of the calling contract. This design principle is crucial for minimizing risks associated with storage context issues that *delegatecall()* might introduce. Ensuring that delegatecall() is only used with contracts declared as libraries is a solid practice.

4.6.4 Example

Listing 10 demonstrates how an NFT fractionalization contract might be exposed to a code injection vulnerability through unsafe use of delegatecall. In this contract, *executeDistributionLogic* can execute arbitrary logic via delegatecall, based on the bytecode provided in data. If *logicContract*

points to a malicious contract, it could lead to unintended alterations in the contract's state, including token ownership manipulation.

```
contract VulnerableNFTFractionalization {
    address public logicContract; // Potentially unsafe external logic contract
    mapping(address => uint256) public ownershipTokens;
    address private owner;

    // Uses delegatecall to execute code from the external contract
    function executeDistributionLogic(bytes memory data) public {
        require(msg.sender == owner, "Not authorized");
        // Unsafe delegatecall allows any code execution from logicContract
        (bool success, ) = logicContract.delegatecall(data);
        require(success, "Execution failed");
    }
}
```

Listing 10: Simplified contract with Code Injection via delegatecall vulnerability

To mitigate the risk of code injection via delegatecall, the contract should strictly control the update of the `logicContract` address and ensure that only verified, safe operations are executable. Also, we introduce a whitelist of approved logic contract addresses and require that any updates to `logicContract` come from this whitelist. Listing 11 illustrates these changes.

```
contract SafeNFTFractionalization {
    address private logicContract; // External contract with verified logic
    mapping(address => uint256) public ownershipTokens;
    address private owner;
    // Authorized operations mapped to their signatures
    mapping(bytes4 => bool) private authorizedOperations;
    mapping(address => bool) public whitelistedLogicContracts; // Whitelist of approved logic contracts


    // Executes only authorized logic via delegatecall
    function executeAuthorizedLogic(bytes memory data) public onlyOwner {
        require(whitelistedLogicContracts[logicContract], "Logic contract not whitelisted");
        require(authorizedOperations[bytes4(data[:4])], "Operation not authorized");
        (bool success, ) = logicContract.delegatecall(data);
        require(success, "Execution failed");
    }
}
```

Listing 11: Simplified contract that mitigates Injection via delegatecall vulnerability

4.7 Randomness Using Block Information

4.7.1 Description

In blockchain-based applications, certain functionalities might require randomness—for instance, distributing a rare NFT fractionally among participants or deciding the winner of a unique NFT in a lottery; block information, such as block hash and block timestamp, can be used to achieve this. This information, however, may be anticipated and slightly modified by miners. When the block timestamp is used as the trigger condition to execute the transaction, it creates a vulnerable situation; dishonest miners can exploit the value of the block timestamp in an unethical manner. This vulnerability was exploited in GovernMental, a Ponzi scheme game [31]. The player who joined the round last and stayed for at least a minute was compensated according to the game rules. A miner who is also a player might change the timestamp to make it look like they were the last to join for more than a minute and, therefore, collect the reward.

4.7.2 Implications on NFT Fractionalization

Randomness Using Block Information could impact Fractionalization solutions. For instance, if randomness derived from block attributes decides the allocation of rare NFT fractions or the winners of rewards, it could lead to outcomes unfairly attributed in favor of those with the ability to influence block information. Similarly, auctions for selling fractionalized NFTs could be manipulated, allowing miners or others with insider advantages to affect the auction's outcome by adjusting the voting period, for example.

4.7.3 Protection Measures

Adopting more reliable sources of randomness is essential to address the vulnerabilities associated with using block information to generate randomness in smart contracts. One solution is employing oracles [73] or other external sources that provide verified randomness robust solution.

Alternatively, cryptographic commitment schemes [74], exemplified by solutions like RANDOA [75], represent another practical approach. These schemes involve participants committing to their inputs in a concealed manner, which are later revealed collectively to generate a random outcome. This method ensures no individual can influence the result based on other participants' commitments, fostering fairness and security.

4.7.4 Example

Listing 12 shows an example of an NFT fractionalization platform that randomly assigns fractional shares of an NFT to participants based on block hash as a source of randomness. This contract is vulnerable because miners, or participants with mining capabilities, can potentially manipulate the block hash to influence the distribution outcome.

```
contract VulnerableFractionDistribution {
    mapping(address => uint256) public fractionsOwned;
    address[] public participants;
    uint256 public totalFractions;

    function distributeFractions() public {
        uint256 blockHashBasedRandom = uint256(blockhash(block.number - 1)) % participants.length;
        address winner = participants[blockHashBasedRandom];
        fractionsOwned[winner] += totalFractions;
    }
}
```

Listing 12: Simplified contract with Randomness Using Block Information vulnerability

To mitigate this vulnerability, the platform can utilize an external oracle to provide a source of verified randomness. This approach ensures that the randomness used for fraction distribution is not predictable or manipulable by miners. Contract illustrated in Listing 13 inherits from VRFConsumerBase, provided by Chainlink [76], to securely request and receive verified random numbers. It uses the Chainlink VRF (Verifiable Random Function) to ensure that participants, including miners, cannot influence randomness in determining the distribution of NFT fractions.

```
contract SecureFractionDistribution is VRFConsumerBase {
    mapping(address => uint256) public fractionsOwned;
    address[] public participants;
    uint256 public totalFractions;
    bytes32 internal keyHash;
    uint256 internal fee;

    // Request randomness
    function getRandomNumber() public returns (bytes32 requestId) {
        require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK");
        return requestRandomness(keyHash, fee);
    }

    // Callback function used by VRF Coordinator
    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal override {
        uint256 randomResult = randomness % participants.length;
        address winner = participants[randomResult];
        fractionsOwned[winner] += totalFractions;
    }
```

Listing 13: Simplified contract with Randomness Using Block Information vulnerability

4.8 Other Vulnerabilities & Summary

Table 4.2 presents the papers that include each vulnerability. We notice that the most mentioned vulnerability is reentrancy, whereas code injection is the least discussed.

The Smart Contract Weakness Classification Registry (SWC) [93] is a community database of known smart contract vulnerabilities; it is often up to date. Each issue includes a description, code samples, and protection measures. Currently, the registry contains 36 vulnerabilities. One has to check this registry often to stay up to speed on the newest threats.

In addition, Rusinek et al. [94] proposed the Smart Contract Security Verification Standard (SCSVS). SCSVS is a 14-point checklist designed to standardize smart contract security for programmers, designers, security auditors, and vendors. By offering recommendations at every level of the smart contract development cycle, SCSVS helps avoid the most known security concerns and vulnerabilities.

	Reentrancy	Front Running	Arithmetic	Mishandled exceptions	Code Injection	Randomness
[28]	X	X	X	X	X	X
[30]	X	X	X	X	X	X
[77]	X					
[78]	X		X			X
[79]	X			X	X	X
[80]	X	X				X
[81]	X					
[82]	X		X	X		X
[31]	X		X	X	X	X
[83]	X	X		X		X
[84]	X			X	X	X
[85]	X	X		X		X
[86]	X	X		X		X
[87]	X					
[4]	X					
[68]	X	X	X			
[88]	X	X		X		
[89]		X				X
[90]			X			
[91]			X			
[92]					X	

Table 4.2: Coverage of vulnerabilities by each reference

4.9 Detection methods

In this section, we present the most common approaches for detecting smart contract vulnerabilities. These include static analysis, dynamic analysis, and formal verification. We will briefly describe each approach and identify its benefits and limitations.

4.9.1 Static Analysis

Static code analysis is a technique of debugging that involves reviewing source code before running it; this technique is also known as white-box testing [95]. It is accomplished by comparing a set of codes to predefined coding rules. There are several techniques to examine static source code; they can be incorporated into a single solution. Compiler technologies are frequently used to develop these techniques, such as Taint Analysis, and Data Flow Analysis [96]. This Section presents some

static analysis techniques, namely Control Flow Graph, Taint Analysis, and Symbolic Analysis.

Control Flow Graph (CFG):

A directed graph that illustrates the control flow of executing source code. Nodes in the graph represent basic blocks—sections of code without jumps—while directed edges depict the transitions from one block to another. An *entry* block has only exit edges, and an *exit* block has only entry edges. For instance, in Figure 4.2, Block 1448 is an entry block, and blocks 1451 and 1452 are exit blocks.

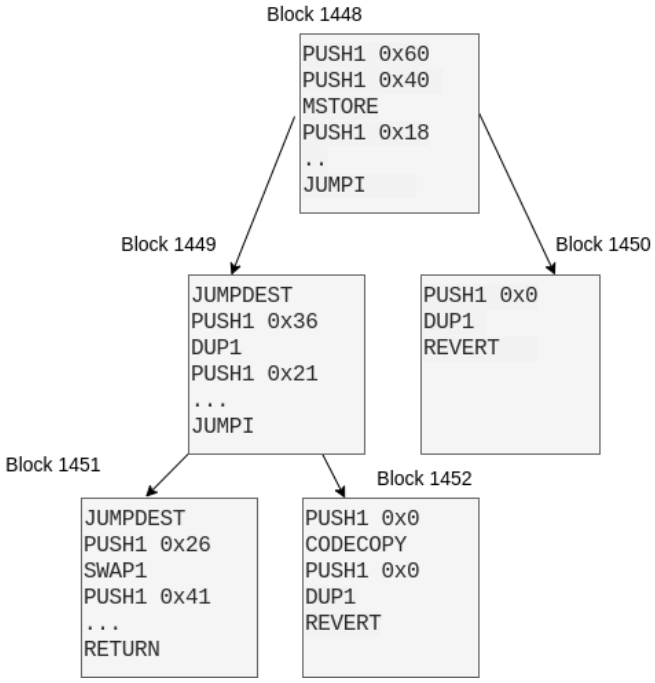


Figure 4.2: Example Control Flow Graph

Figure 4.3 demonstrates the standard process for creating a CFG in the context of Ethereum, starting with **Parsing bytecode**, where the compiler version is extracted from the metadata and the remaining bytecode is parsed into opcodes (e.g., 0x04 represents DIV). The second step, **Identification of basic blocks**, involves defining blocks as a sequence of opcodes running successively from a jump target to a jump instruction. Basic blocks are delineated by their offset in the bytecode. The final step involves **computing the edges** by identifying the destination offsets, with special attention to different types of jumps like *pushed jumps* and *orphan jumps*, which are crucial for the graph's completeness.

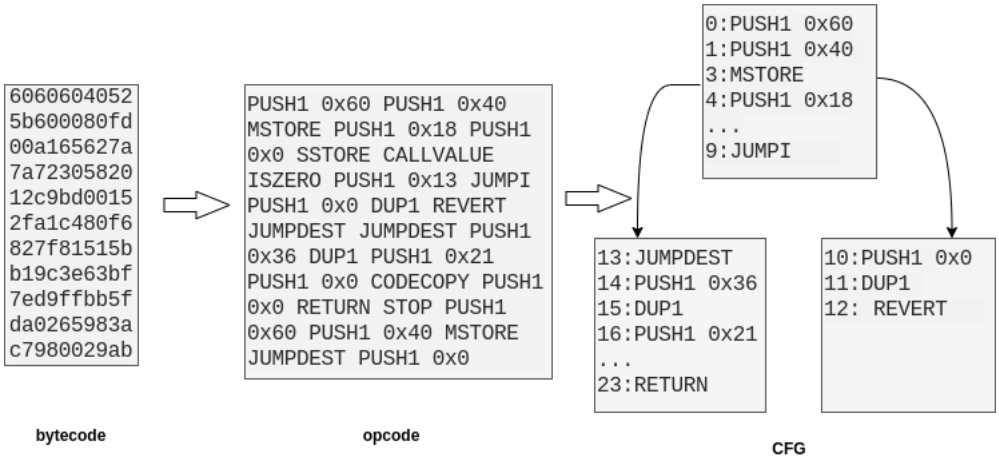


Figure 4.3: Process of generating CFG

Operation	Opcode	Description
Flow Operations	JUMP	Alters the program counter
	JUMPI	Conditionally alters the program counter
	JUMPDEST	Marks a valid destination for jumps
System Operations	STOP	Halts execution
	REVERT	Halts execution reverting state changes but returning data and remaining gas
	RETURN	Halts execution returning output data
	INVALID	Designated invalid instruction
	SELFDESTRUCT	Halts execution and registers account for later deletion

Table 4.3: Opcodes that alter execution

Taint Analysis:

This analysis detects input variables from untrusted sources that could be manipulated by an attacker (e.g., environmental data or function parameters). Tagged as *tainted*, these inputs are traced to sensitive functions, known as *sinks*. If a tainted variable reaches a sink, it is flagged as a potential vulnerability. In Ethereum smart contracts, taint analysis typically follows CFG generation, using opcodes that could be exploited by an attacker for data insertion.

An example of taint analysis in action is the detection of arithmetic bugs by tracing inputs like CALLDATALOAD and SLOAD through to operations such as ADD, MUL, and SUB without a safeguard mechanism.

Symbolic Analysis:

Operation	Opcode	Description
Block Information	GASLIMIT TIMESTAMP	Gets the current block gas limit Gets the current block timestamp
Environment Information	CALLER CALLDATALOAD CALLDATACOPY	Gets caller address Gets input data of current environment Copies input data in current environment to memory
Flow Operations	SLOAD MLOAD	Loads word from storage Loads word from memory

Table 4.4: Opcodes that allow data insertion

Symbolic execution abstracts program execution to span multiple pathways, using *symbols* as inputs and expressing outputs based on these symbols. Symbolic paths each have a condition based on constraints that must be met for execution to proceed. If the condition is unsatisfiable, the path is deemed infeasible; otherwise, it is feasible. In finding exploits, most tools use the Satisfiability Modulo Theories (SMT) solver to check if symbolic outputs are feasible or provide counterexamples.

4.9.2 Dynamic Analysis

The term "dynamic analysis" refers to observing code while it is executed in its original context. It is also known as black-box testing because it is performed without access to the source code. It works the same way as an attacker who feeds malicious code or unpredictable input to the appropriate functions of a program to look for vulnerabilities. The most common technique is Fuzzing.

Fuzzing:

Fuzzy testing, often known as fuzzing, is a kind of automated software testing that involves injecting incorrect, malformed, or unpredictable inputs into a system; the objective is to uncover software flaws and vulnerabilities. A fuzzing tool injects these inputs into the program and then watches for problems such as crashes or data leaks; it also can perform static analysis on execution traces. In the case of smart contracts, Wang et al. [97] deploy the smart contract on a test network; then, they monitor the balance of the smart contract to identify basic misappropriation. This technique removes the necessity for particular patterns to determine a vulnerability.

4.9.3 Formal Verification

Formal verification is a technique that automates bug detection of a hardware or software system by comparing a formal system model to formal requirements or behavior specifications. Through mathematical analysis, formal verification can provide a high level of confidence.

To conduct a formal verification, we need to provide program specifications. A program specification defines the purpose of the program and the scenarios that are allowed or not to be executed. Tools usually build deduction trees to verify a property where the root presents a Hoare triplet [98]. A Hoare triplet consists of (a) *Pre condition*: It is the initial state; (b) *Instructions*: A series of transactions; and (c) *Post condition*: It is the property to verify. To build deduction trees, proving algorithms are usually used, such as Coq [99] and Isabelle/HOL [98].

There are a few contributions that propose languages to write formal specifications. Permenev et al. [100] proposed a specification language called VerX, where the syntax is similar to Solidity; it supports temporal logic. The Ethereum community also proposed a language named *Act* [101], which is a specification for Ethereum Virtual Machine (EVM) programs.

4.9.4 Comparison

The goal of using detection methods such as static, dynamic, and formal verification is to ensure that smart contracts are correctly executed in all states; this is to ensure that no vulnerabilities are produced and that specifications are respected. Table 4.5 shows the papers that discuss each methodology.

Static code analysis tools can potentially produce false negative results; vulnerabilities occur but are not reported. This could happen because the analysis tool doesn't allow for many test scenarios; it doesn't go into detail when looking at different states because it usually gives you a timeout. Various vulnerabilities may be false negatives in static analysis; however, they can be detected correctly utilizing dynamic analysis tools. Thus, it is strongly recommended that both types of analysis be combined.

Dynamic analysis generates transactions with random inputs, which means that it observes random states depending on the inputs. The challenge for these techniques is to generate enough transactions and carefully select inputs to achieve maximum coverage.

Formal verification can provide full coverage for the specification under consideration because it employs mathematical analysis. This guarantees the satisfaction of a property if verified; however, in general, we cannot cover everything because the verification can become quickly intractable. Thus, false negatives can still be present. In addition, formal verification requires a skilled programmer who can express a smart contract as a mathematical high-level specification while accounting for a specific low-level virtual machine. This operation takes a long time and requires a lot of resources. Audit firms usually make use of formal verification [102].

Vulnerabilities	References
Static analysis	[28] [30] [80] [86] [90] [88] [92] [4] [103] [68] [91] [100] [104]
Dynamic analysis	[28] [30] [105] [106] [79] [107] [108] [109] [110]
Formal verification	[28] [30] [111] [103] [98] [80] [104]

Table 4.5: Detection methods References

4.10 Vulnerability Detection Tools

This section will cover the most popular tools for detecting smart contract vulnerabilities. To investigate available tools, we first looked through academic literature and review articles (see Table 4.6). Then, we covered five of the most commonly used tools, as indicated by developer forums, and the number of forks and update frequency on GitHub.

4.10.1 Oyente

- **Description:** Luu et al. introduced Oyente [86], the first symbolic execution tool designed for analyzing Ethereum smart contracts. Recognized as a foundational tool in the domain, Oyente has significantly influenced the development of subsequent tools, such as Maian [105] and Osiris [90]. The architecture of Oyente comprises four key components, each contributing to its comprehensive analysis capabilities:

#	Tool	Detection Technique	Last update
1	Slither [4] [112]	Static Analysis	May 2023
2	MythX [113]	static, dynamic & Symbolic Execution	Not Public
3	Mythril [114] [115]	Symbolic Execution	May 2023
4	Echidna [106] [116]	Fuzzer	Apr 2023
5	Manticore [110] [117]	Symbolic Execution	June 2022
6	Securify [88] [118]	Static Analysis	Sep 2021
7	KEVM [103] [119]	Static Analysis	Apr 2022
8	Smartcheck [68] [120]	Static Analysis	Dec 2019 deprecated
9	MadMax [91] [121]	static analysis	Jun 2021
10	Vertigo [122] [123]	Mutation Testing	Feb 2021
11	EtherSolve [87] [124]	CFG Extraction	Nov 2021
12	Octopus [125]	Symbolic Execution	Nov 2020
13	Oyente [86] [126]	Symbolic Execution	Nov 2020 deprecated
14	ERC20 Verifier [127]	Verify ERC20 Compatibility	Nov 2019
15	Solgraph [128]	CFG Extraction	Jan 2019
16	Osiris [90] [129]	Symbolic Execution	Sep 2018

Table 4.6: Ethereum Smart Contract Vulnerability Detection Tools

(a) *CFGBuilder*: This component is responsible for constructing the Control Flow Graph (CFG), enabling a structured representation of all possible execution paths.

(b) *Explorer*: The Explorer component processes the Ethereum state as input. It operates through a loop that executes instructions based on the current state, producing a symbolic execution trace. This process continues iteratively until all states are exhausted or a predetermined timeout is reached. The Explorer utilizes the Z3 SMT solver [130] to identify infeasible execution

paths, enhancing the precision of its analysis.

(c) *CoreAnalysis*: Focused on vulnerability detection, CoreAnalysis scans the symbolic trace for patterns indicative of known vulnerabilities. This targeted approach allows Oyente to efficiently identify potential security issues within the smart contract code.

(d) *Validator*: To minimize false positives and ensure the reliability of its findings, the Validator component cross-references the identified vulnerabilities against the Z3 SMT solver. This step verifies the accuracy of flagged vulnerabilities, providing users with validated results.

Oyente detects seven types of vulnerabilities, namely Re-entrancy, Integer overflow/underflow, Transaction order dependence, Timestamp dependence, Callstack Depth, EVM Code Coverage, and Parity Multisig bugs.

- **Execution Example:** For evaluating Oyente's capabilities in the context of smart contracts dealing with token sales and potentially fractionalized assets, we utilize the *TokenSaleChallenge.sol* smart contract, sourced from the Capture the Ether challenge series [131]. The contract embodies a token sale mechanism where tokens are bought and sold at a fixed price, presenting an analogous scenario to NFT fractionalization where individual tokens could represent shares of a fractionalized NFT.

  The primary aim is to assess Oyente's effectiveness in detecting vulnerabilities that could impact contracts handling fractionalized NFTs. The results, depicted in Figure 4.4, highlight an Integer Overflow vulnerability and the execution trace where this issue is detected.

```
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.21, The latest supported version is 0.4.19
INFO:root:contract tokensalechallenge.sol:TokenSaleChallenge:
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                    94.8%
INFO:symExec:     Integer Underflow:                    False
INFO:symExec:     Integer Overflow:                     True
INFO:symExec:     Parity Multisig Bug 2:                False
INFO:symExec:     Callstack Depth Attack Vulnerability:  False
INFO:symExec:     Transaction-Ordering Dependence (TOD): False
INFO:symExec:     Timestamp Dependency:                 False
INFO:symExec:     Re-Entrancy Vulnerability:            False
INFO:symExec:tokensalechallenge.sol:18:9: Warning: Integer Overflow.
        balanceOf[msg.sender] += numTokens
Integer Overflow occurs if:
    balanceOf[msg.sender] = 1
    numTokens = 1157920892373161954235709850086879078532699846656405640394575840079131296399335
INFO:symExec:   ====== Analysis Completed ======
```

Figure 4.4: Oyente Output

- **Pros and Cons:** Oyente's deployment via a Docker image significantly simplifies its setup process, making it accessible for users with varying expertise in blockchain technology. This

ease of use facilitates quick integration into development workflows, allowing for the immediate analysis of Ethereum smart contracts. Despite its user-friendly setup, Oyente's compatibility is somewhat limited; it supports Ethereum compiler versions only up to 0.4.17. This restriction may hinder its applicability to smart contracts compiled with newer versions of the Solidity compiler, potentially limiting its utility in analyzing the latest smart contract developments.

The output provided by Oyente is clear and concise, enabling developers and auditors to interpret vulnerability reports easily. Moreover, Oyente has played a pivotal role in evolving smart contract analysis tools. It has laid the groundwork for subsequent innovations, including Maian [132] and Osiris [90]. These tools have built upon Oyente's foundational techniques, such as constructing Control Flow Graphs (CFGs), to enhance accuracy in detecting vulnerabilities and orphan paths within smart contracts.

4.10.2 Slither

- **Description:** Feist et al. introduced Slither [4], a comprehensive static analysis framework designed to identify vulnerabilities, optimize code performance, and deepen the understanding of smart contract codebases. The procedure it employs is in multiple stages:

  (a) *Data Retrieval*: This initial phase uses the Solidity compiler to construct the Abstract Syntax Tree (AST) of a smart contract's source code. This step not only provides a detailed overview of the contract's structure, including its Control Flow Graph (CFG) and inheritance relationships, but also sets the foundation for deeper analysis.

  (b) *Conversion into SlitherIR*: Following the retrieval of structural data, Slither converts the smart contract source code into its proprietary Intermediate Representation (SlithIR). SlithIR facilitates a more nuanced and comprehensive analysis by abstracting the code into a form that's easier for Slither's analysis algorithms to process.

  (c) *Examination of the Code*: In the final stage, Slither thoroughly examines the smart contract code. This examination includes tracking the read and write operations on variables, identifying their types, and detecting potentially unsafe functions that malicious addresses could exploit to perform high-level operations. Additionally, Slither assesses data dependencies, flagging variables

influenced by external user input as tainted.

Slither can identify over 70 distinct types of bugs, including those that could lead to the self-destruction of smart contracts, code injection via delegatecall, frozen ether, and reentrancy attacks [133].

- **Execution Example:**

  Using Slither to identify potential security vulnerabilities, we analyzed the *FractionalNFTMarket* smart contract in Listing 14. The analysis results are presented in Figure 4.5. Slither has successfully detected a reentrancy vulnerability within the *sellShares* function of the contract, which could allow an attacker to exploit the contract by recursively calling the function to drain its funds. In addition to detecting this critical vulnerability, Slither offered recommendations for code improvements and provided a comprehensive set of information, including a human-readable summary, an inheritance graph, and the Control Flow Graph (CFG) of each function.

```
1
2  contract FractionalNFTMarket {
3      address public nftOwner;
4      uint256 public totalShares = 100;
5      mapping(address => uint256) public sharesOwned;
6      uint256 public pricePerShare = 0.01 ether;
7
8      constructor() {
9          nftOwner = msg.sender;
10         sharesOwned[nftOwner] = totalShares;
11     }
12
13     function sellShares(uint256 shares) public {
14         require(sharesOwned[msg.sender] >= shares, "Not enough shares owned");
15
16         (bool sent, ) = msg.sender.call{value: shares * pricePerShare}("");
17         require(sent, "Failed to send Ether");
18
19         sharesOwned[msg.sender] -= shares;
20         sharesOwned[nftOwner] += shares;
21         emit ShareSold(msg.sender, shares);
22     }
23
24 }
25
```

Listing 14: FractionalNFTMarket Smart Contract

Slither also provides helpful information, including a human-readable summary of each function's scanned contract, inheritance graph, and CFG; the complete list can be found in the GitHub repository [112].

Figure 4.5: Slither Analysis Output for FractionalNFTMarket Contract

- **Pros and Cons:** Slither, an open-source static analysis tool, is recognized for its efficiency in auditing smart contracts. It is user-friendly, offering straightforward installation options via Docker or Python package managers. With the capability to detect approximately 70 types of vulnerabilities, Slither facilitates improved code understanding through visual aids like contract graphs. Moreover, it examines smart contract compliance with established Ethereum Request for Comments (ERC) standards, including ERC-20 and ERC-777 [134].

  One limitation of Slither is its tendency to report false positives, which can complicate the auditing process by requiring additional validation to review the findings. Crytic [135], a premium service, can be utilized for a more comprehensive analysis. Crytic extends Slither's capabilities by identifying 50 types of faults that Slither might miss. Furthermore, Crytic offers integration with GitHub, allowing for automated testing on pull requests, thereby enhancing the development workflow and ensuring continuous contract integrity.

4.10.3 Mythril

- **Description:** Mythril, implemented by Durieux et al. [114], is a static analysis tool designed to detect potential vulnerabilities in smart contracts. It uses symbolic execution to analyze the EVM bytecode of contracts, enabling it to simulate various execution paths and identify possible security flaws. Mythril offers an interactive Python-based command line, allowing users to closely inspect contract behavior.

  The tool systematically generates a Control Flow Graph (CFG) to visualize possible execution

pathways and employs symbolic execution to limit the scope of its analysis to the most relevant paths. Mythril allows for practical testing of smart contract resilience by generating concrete input values that can potentially trigger vulnerabilities. Central to its operation is the Z3 SMT solver [130], which Mythril utilizes to assess the feasibility of execution paths and to solve complex constraints that arise during symbolic execution

Mythril's detection capabilities encompass a wide array of vulnerabilities, which include, but are not limited to, issues like Untrusted Delegate Call, Predictable Variable Dependence, Deprecated Opcode Usage, Ether Theft, Exception Handling Errors, Unsafe External Calls, Integer Overflow and Underflow, Repeated Transfers, Contract Self-Destruction, External Calls Leading to State Changes, Unchecked Return Values, User-Defined Assertions, Arbitrary Storage Writing, and Unconstrained Jumps [136].

- **Execution Example:** The *FractionalNFTMarket.sol* smart contract, presented in Listing 14, has been analyzed using Mythril to uncover potential security issues. The resulting output is depicted in Figure 4.6. Mythril has identified a reentrancy vulnerability within the contract's 'sellShares' function. The output provides detailed information, including the state of the contract, when the vulnerability can be triggered, and the specific transaction sequence that leads to the issue. Unlike Slither, Mythril offers not only the identification of vulnerabilities but also the concrete inputs that cause them, enhancing the developer's understanding of the contract's weaknesses. Nevertheless, Mythril's output does not extend to include recommendations for remediation of the detected issues.

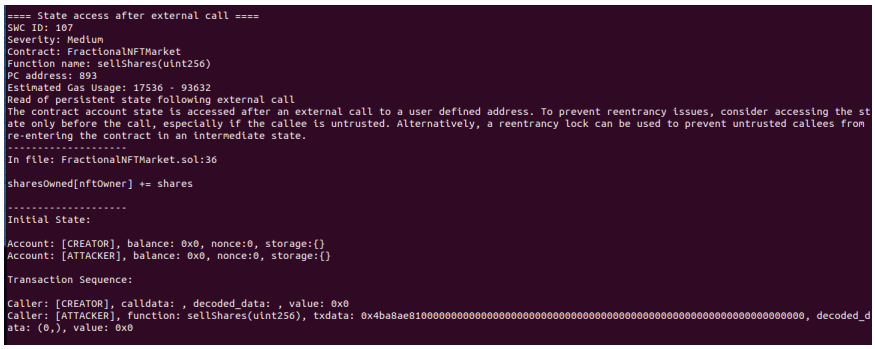


Figure 4.6: Mythril Output

- **Pros and Cons:** Mythril stands out for its accessibility. It offers straightforward installation

options via Docker or Python package manager, which streamlines the setup process. It allows for granular analysis, with the flexibility to test individual functions through custom Python scripts. This targeted testing can be particularly advantageous for developers seeking to debug specific aspects of their smart contract code.

However, Mythril's thorough approach to smart contract analysis can be computationally intensive, often resulting in longer analysis times than tools like Slither. The resource-heavy nature of Mythril's symbolic execution process may not be the most efficient choice for rapid iteration during development. For those seeking a more advanced feature set and cloud-based capabilities, Mythril's commercial counterpart, MythX [113], offers enhanced analysis power and can be seamlessly integrated into continuous integration/continuous deployment (CI/CD) pipelines [137].

4.10.4 Manticore

- **Description:** Manticore, developed by Mossberg et al. [110] and maintained by Trail of Bits [138], is an analysis tool for smart contracts. It is designed to identify vulnerabilities by employing symbolic execution to explore various execution paths within the Ethereum Virtual Machine (EVM) bytecode. Utilizing the Z3 SMT solver [130], Manticore efficiently generates inputs that trigger different computation paths and records the resulting execution traces.

  The tool is particularly adept at providing detailed insights into the smart contract's behavior. It translates Solidity code into EVM bytecode, which is then analyzed to uncover potential security flaws. Among the vulnerabilities Manticore can detect are delegatecalls, arithmetic overflows and underflows, reentrancy, unsafe or manipulable instructions, and reachable self-destruct operations [139]. These vulnerabilities are reported about the original Solidity source code, enhancing the developer's ability to address the identified issues.

- **Execution Example:** For our examination of Manticore, we used the *TokenSaleChallenge.sol* [131], the same one used to test Oyente. This contract represents a simplified model for trading tokens, which can be seen as stand-ins for fractional tokens of an NFT.

  The tool is extremely slow as it takes more than one hour to perform the test; the other tools take less than a minute to use the same smart contract and machine. The analysis output is a

folder containing a summary report, traces, and analysis for each test case. Manticore generated 51 test cases but failed to detect the smart contact's reentrancy vulnerability, as shown in Figure 4.7.

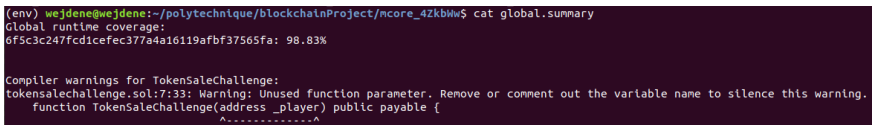


Figure 4.7: Manticore Output

- **Pros and Cons:** Manticore offers a diverse set of command-line tools along with scriptable Python APIs, suitable for a variety of use cases and allowing for in-depth analysis procedures. It covers a broad spectrum of known vulnerabilities, making it a valuable asset for developers seeking to secure their smart contracts.

  However, the detailed nature of Manticore's analysis comes with a trade-off in terms of performance. The tool can take significantly longer to complete its analysis than others, sometimes leading to timeouts. This is often due to its comprehensive symbolic execution approach, which, while powerful, is also resource-intensive. Additionally, Manticore requires considerable memory, which could be a limiting factor for users with constrained hardware resources.

4.10.5 Echidna

- **Description:** Echidna, implemented by Grieco et al. [106], is a fuzzing tool for testing Ethereum smart contracts. Drawing from the principles established by QuickCheck [140], Echidna focuses on property-based testing where it aims to falsify user-defined invariants that represent potential errors of the contract under test [141]. These invariants, known as *echidna properties*, are Solidity functions without arguments that should return "true" to indicate success. The functions are conventionally prefixed with *echidna_* to denote their purpose as test properties.

  When Echidna generates transactions that cause any of these properties to return "false" or trigger an error, it reports them, effectively highlighting flaws within the contract [142]. Echidna utilizes techniques to generate these inputs, including iterative feedback loops, structural heuristics, and variations on known inputs. This approach enables Echidna to uncover bugs that may

not be apparent through traditional testing methods.

- **Execution Example:** To test Echidna, we have adapted the *tokensalechallenge.sol* smart contract [131]. We created a derived contract *TestTokenSaleChallenge* that extends the original *TestTokenSaleChallenge* contract shown in Listing 15. The test properties in the *TestTokenSaleChallenge* are designed to check for arithmetic errors that can occur in the logic used for NFT fractionalization.

  The *echidna_test_overflow* function aims to validate that token balances do not exceed the maximum uint256 value, a critical check to prevent overflow in token allocations, which could mirror similar concerns in the distribution of fractional NFT shares. Conversely, *echidna_test_underflow* ensures that the balance never drops below zero.

  By defining these test functions, Echidna will attempt to generate test inputs that trigger these conditions to ensure the contract behaves as expected.

```
1
2  pragma solidity ^0.5.0;
3
4  import "./TokenSaleChallenge.sol";
5
6  contract TestTokenSaleChallenge is TokenSaleChallenge {
7
8
9      function echidna_test_overflow() public view returns (bool) {
10         uint256 MAX_UINT = ~uint256(0);
11         uint256 balance = balanceOf[msg.sender];
12         return balance < MAX_UINT;
13     }
14
15     function echidna_test_underflow() public view returns (bool) {
16         uint256 balance = balanceOf[msg.sender];
17         return balance > 0;
18     }
19 }
20
21
```

Listing 15: TestTokenSaleChallenge Smart Contract

Echidna generates its results directly to the console, as figure 4.8 shows, distinguishing between successfully validated and failed properties. Echidna offers a counterexample for each property that didn't meet the validation criteria, which is essentially a step-by-step breakdown of how the property failed under specific conditions. In our example, the underflow test did not pass, even without initiating a transaction. This was because the `msg.sender` was left empty, indicating

that the system can encounter failures in scenarios where sender information is not provided. However, Echidna didn't detect an overflow issue, possibly due to a short timeout period, which prevented it from producing a sequence revealing this vulnerability as we did not change the default configuration.

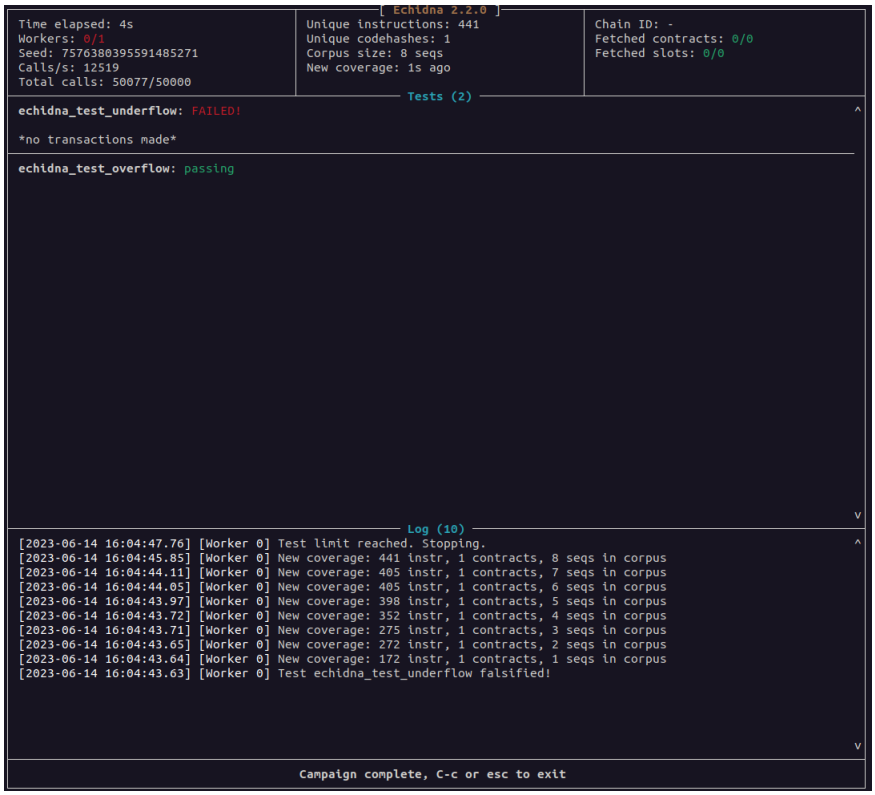


Figure 4.8: Echidna Output

- **Pros and Cons:** One of Echidna's notable strengths is its straightforward setup process. It offers users multiple installation options, including direct build with Stack, Docker, or Homebrew. This flexibility facilitates accessibility across different platforms and development environments.

  However, users are required to define the properties they wish to test explicitly. This feature allows for customized and focused testing scenarios, ensuring the analysis is directly relevant to the contract's intended behaviors and security assumptions. Despite the advantage of tailored testing, this approach adds a layer of complexity. Users must thoroughly understand their contract's logic and potential vulnerabilities to define meaningful test properties effectively. This necessity for manual property definition demands a higher level of engagement and expertise from the user than tools that automatically infer test cases or vulnerabilities.

4.10.6 Summary

Table 4.7 presents a comparative analysis of Oyente, Slither, Mythril, Manticore, and Echidna. The comparison is based on several criteria, including the number of detected vulnerabilities, the methodology employed, whether the tool originates from academic research or a company, the level of code analysis (EVM bytecode vs. Solidity source code), the required version of Solidity, and the extent of available documentation.

	Oyente	Slither	Mythril	Manticore	Echidna
Type of detected vulnerabilities	4	70	14	10	-
Methodology	Symbolic	Symbolic	Static	Symbolic	Fuzzing
Code Level	EVM	Solidity	Solidity	EVM	Solidity
Restrictions (Solidity Version)	≤ 0.4	≥ 0.4	≥ 0.4	≥ 0.4	not specified
Documentation	Low	Medium	High	High	Medium

Table 4.7: Qualitative comparison of selected tools

The selection of an appropriate analysis tool is critical in the context of NFT fractionalization. These contracts often involve complex interactions and financial transactions, making them prime targets for security vulnerabilities that could compromise the integrity of the fractionalized assets.

Static analysis tools like Slither, Mythril, and Manticore offer comprehensive means to identify a broad range of vulnerabilities in Solidity code. Oyente, with its focus on EVM bytecode, provides an additional layer of analysis, especially for contracts compiled from versions of Solidity up to 0.4. While not directly listing detected vulnerabilities, Echidna's fuzzing approach is invaluable for testing the robustness of smart contracts against unexpected inputs and states, a crucial aspect when dealing with the interactive nature of fractionalized NFT platforms.

In summary, static analysis tools are widely used because they are straightforward and efficient. However, choosing the best tool also depends on the complexity of the contract and the developer's skills, given that fuzzing requires a lot of resources and formal verification requires specialized knowledge. A combination of these tools might provide the most comprehensive security assessment for NFT fractionalization, ensuring the reliability and security of fractionalized assets in the blockchain ecosystem.

4.11 Guidelines for Secure Smart Contracts

Ensuring the security of smart contracts is paramount, mainly when they are used for NFT fractionalization, where vulnerabilities can lead to significant financial losses. The steps in this section, summarized in Figure 4.9, offer a set of guidelines for creating robust smart contracts.

Visualization is the first critical step, offering a global view of the contract's flow and interaction between its components. This is crucial in complex contracts like those used for fractionalizing NFTs. Tools like *Slither* and *Solidity Visual Developer* can help spot critical functions that handle the division and ownership of fractional NFT shares.

The next step involves automated analysis for common bug detection. Combining *Mythril* and *Slither* is advisable for a broad vulnerability assessment. For targeting specific functions, for example, that manage fee distribution or collection, *Manticore* is beneficial as it allows for the simulation of complex use cases often seen in NFT platforms. Dynamic analysis with *Echidna* plays a significant role in testing the reliability of contract functions under unpredictable conditions.

For contracts involving NFT fractionalization, ensuring compliance with relevant standards and testing unique features such as token transfer mechanisms and AMM is crucial. *Slither's* functionalities can be employed to verify adherence to standards like ERC-721 for NFTs and ERC-20 for fungible token fractions.

Echidna and *Manticore* can be directed to test custom security properties defined in Solidity. Such properties are essential for capturing the specific requirements of fractional ownership, including the fair distribution of fees, setting appropriate withdrawal limits, and safeguarding ownership rights. Additionally, *Slither* complements these tools by using its Python API to conduct targeted validation of security concerns, such as verifying the control mechanisms over ownership and ensuring the traceability of shares.

Although challenging, formal verification becomes crucial for contracts handling fragmented ownership of high-value NFTs. This step confirms that the contract's logic aligns with the intended fractionalization behavior.

Lastly, manual verification of the automated tools' output ensures that the nuances of NFT fractionalization are appropriately considered and that identified issues are contextualized within the contract's framework. Corrective measures should be based on established security patterns and, once applied, should be followed by a re-run of the previous steps to confirm the resolution of the issues.

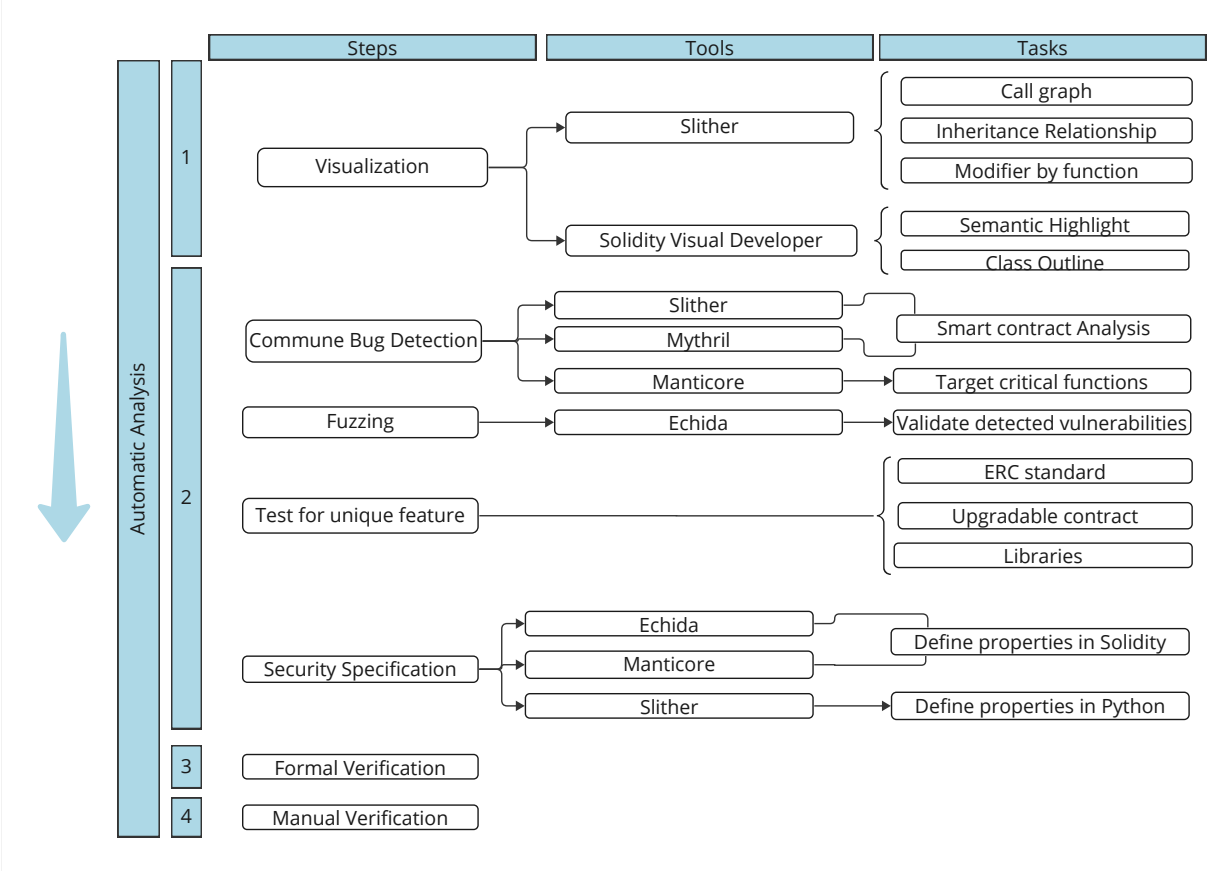


Figure 4.9: Auditing Process for Smart Contracts

Chapter 5:

Use Case

This chapter examines the application and security of current fractionalization solutions by analyzing their audit reports. We also present a use case involving a simple solution that implements the proposed standard, analyzing its security mechanisms to highlight the practical implications of the research findings. This chapter aims to bridge the gap between theoretical proposals and their practical implementation, showcasing fractionalized smart contracts' real-world applicability and security considerations.

5.1 Security Analysis of Current Solutions

5.1.1 Security Analysis

We have conducted a technical security analysis on Tessera, NFTX, and Unicly's smart contracts. We used the Slither and Mythril tools to detect vulnerabilities in smart contracts. The analysis predominantly revealed outdated syntax within the contracts without uncovering critical security flaws. Notably, Slither flagged a potential reentrancy vulnerability in Unicly's contract. However, further examination classified this as a false positive.

All three platforms have undergone external security audits, a critical step in validating the integrity and security of smart contracts. These audits, conducted by independent third parties, identify vulnerabilities that automated tools might miss and suggest mitigation strategies.

5.1.2 Audit Report Study

We conduct a comparative analysis of audit findings from three blockchain projects: Tessera, Unicly, and NFTX. By examining the insights provided by audit reports, we aim to reveal common vulnerabilities in NFT fractionalization platforms.

- **Tessera:** Audited by **Code4rena (C4)** [143], a collective of security experts including researchers, auditors, and developers specializing in smart contract security. Code4rena employs a unique audit contest model, engaging community members to identify smart contract vulnerabilities for bounties. The comprehensive audit by C4 revealed a total of 32 issues, categorized into 20 high-risk, 112 medium, and 97 low or informational [56].
- **Unicly: Quantstamp** [144], a leader in blockchain security, conducted the audit for Unicly. Leveraging advanced tools such as computer-aided reasoning, Quantstamp's team, with extensive experience in formal verification and software security, aims to enhance blockchain adoption through rigorous security assessments. Their audit report for Unicly noted 16 issues: 1 high-risk, four medium and 11 low or informational [57].
- **NFTX:** NFTX's security assessment was undertaken by two firms: **Trail of Bits** and **SECBIT** [145]. Trail of Bits [145], known for its high-end security research and real-world attacker mentality, aims to reduce risk and strengthen code integrity. Their audit reported 15 issues, with a breakdown of 0 high-risk, one medium, and 14 low or informational [59]. SECBIT [146], focusing on formal verification and blockchain security, uncovered ten issues, four being medium and six informational [58].

For C4 [56], the severity of vulnerabilities is assessed using a methodology aligned with OWASP standards [147]. Whereas Quantstamp, Trail of Bits, and SECBIT employ a similar but distinct criterion for categorizing risk [57] [59] [58] :

High Risk: Issues that could significantly impact a vast user base, harm the client's reputation, or lead to substantial financial loss. This includes vulnerabilities that compromise the integrity of contracts, enabling theft or asset locking.

Medium Risk: Vulnerabilities that affect a subset of users, potentially harm the client's reputation or result in moderate financial impacts. These issues may also impair contract security under certain conditions, affecting stakeholder benefits.

Low Risk: Minor vulnerabilities unlikely to be exploited frequently or cause no actual harm to the contract's functionality.

- **By Vulnerabilities type:** Table 5.1 offers a comprehensive analysis of vulnerabilities identified in the smart contracts Tessera, Unicly, and NFTX, each categorized by its nature. Tessera displays a broad and severe spectrum of vulnerabilities, particularly in logic flaws, authorization, and arithmetic operations. This extensive vulnerability range could be attributed to Tessera's approach of using migrations to update vault and governance mechanisms. Such a method, involving asset migration to new vaults with governance updates, inherently increases risk exposure. To mitigate these risks, Tessera has reevaluated its governance update mechanisms and asset migration strategies, ensuring tighter security protocols are in place.

  Unicly's audit reveals concentrated concerns primarily around medium-severity logic flaws. They have addressed them through targeted security enhancements mostly related to logic implementation. Meanwhile, NFTX had a vulnerability to front running, in addition to medium-severity authorization issues. To solve this, NFTX has refined its transaction ordering and access control systems to safeguard against potential exploits.

  Logic flaws emerge as the most common type of vulnerability across the platforms, underscoring the complexity of detecting such issues that require a comprehensive understanding of the platform's intended functionality. These vulnerabilities often necessitate extensive manual review and unit testing for detection. Additionally, access control vulnerabilities represent a significant concern in fractionalization, where precise role definitions are crucial to prevent unauthorized access and potential asset theft.

- **By Audit Tools:** Table 5.2 offers an overview of the audit tools employed during the security evaluations of Tessera, Unicly, and NFTX.

  Tessera used development and analysis tools such as Hardhat, Foundry, and Solidity Visual Developer for VSCode, which are more focused on development environments and code quality

Vulnerabilities	**Tessera**	**Unicly**	**NFTX**
Reentrancy	2 High	0	0
Front running	0	0	1 High
Arithmetic	3 High, 1 Medium	0	0
Mishandled Exception	1 High, 1 Medium	0	0
Code injection	1 High, 1 Medium	0	0
Randomness using block info	0	0	0
Logic Flaw	9 High, 3 Medium	4 Medium	1 Medium, 2 Low
Authorization or Access Control	3 High, 2 Medium	1 High, 1 Low	5 Medium
Validation Error	1 High, 3 Medium	2 Low	2 Low

Table 5.1: Comparison of vulnerability types across Tessera, Unicly, and NFTX.

improvements. The choice of tools suggests a focus on internal development practices and code verification rather than external vulnerability scanning or automated security checks. Notably, Tessera's audit was conducted by the community rather than a professional audit firm, reflecting the common tools favored within the community.

Unicly shows a selective approach by utilizing Slither, indicating a reliance on this specific tool for identifying security vulnerabilities. Slither is known for its effectiveness in static analysis and detecting common smart contract vulnerabilities, which suggests Unicly's audit was streamlined towards leveraging Slither's capabilities for a targeted security review.

NFTX audit by Trail of Bits displays a broader and more varied toolkit. Trail of Bits' audit incorporated Slither, Echidna, and Manticore, indicating a comprehensive approach using static analysis, fuzz testing, and symbolic execution to uncover vulnerabilities. Whereas SECBIT utilized a mix of public tools like Slither, SmartCheck, Mythril, and Securify, along with internal tools (adelaide, sf-checker, and badmsg.sender), showcasing a hybrid approach that combines established analysis techniques with proprietary methodologies for a tailored audit experience.

The analysis reveals the importance of audits in enhancing the reliability and security of smart contracts. This comparative analysis underscores the need for ongoing research and development in smart contract security to address emerging vulnerabilities and threats.

Tools	Tessera	Unicly	NFTX - Trail of Bits	NFTX -SECBIT
Slither		x	x	x
Echidna			x	
Manticore			x	
SmartCheck				x
Mythril				x
Securify				x
Hardhat	x			
Foundry	x			
Solidity Visual Developer of VSCode	x			
adelaide (internal)				x
sf-checker (internal)				x
badmsg.sender (internal)				x

Table 5.2: Audit tools used in Tessera, Unicly, and NFTX audits.

5.2 Design of concrete Implementation of NFT fractionalization

5.2.1 Methodology

In our implementation of the proposed standard, as detailed in Section 3.3, we began by visualizing the interactions between smart contracts for different scenarios using sequence diagrams.

During the implementation phase, we chose Truffle [148] as our development environment, which offers a complete suite for implementing and testing smart contracts. After refining our contracts, we deployed them on Truffle's testnet to conduct manual testing. This crucial step allowed us to verify the functionalities of our smart contracts and to simulate real-world operations without the financial risks associated with the Ethereum mainnet.

To complement our manual testing efforts, we conducted automated security assessments. Starting with Slither, we screened for security vulnerabilities previously discussed in Chapter 4. This was followed by a more thorough investigation using Mythril, enhancing our testing and validation process. We also utilized Echidna for state-of-the-art fuzz testing, focusing on the more nuanced aspects of smart contract integrity.

Although we did not include formal verification in our testing protocol due to its complexity, our

comprehensive, multi-layered approach ensured robust validation of our proposed implementation

5.2.2 NFT Deposit and Fractional Token Minting

The sequence diagram Figure 5.1) present a comprehensive visual representation of the NFT deposit and fractional token minting process within our proposed NFT fractionalization solution.

Preconditions and Initial Setup: Prior to the sequence initiation, the NFT Owner must possess an ERC-721 or ERC-1155 compliant NFT, and the relevant smart contracts are assumed to be deployed and operational on the Ethereum blockchain. The Fractional Token Contract is based on the ERC-20 token standard, supporting token minting capabilities.

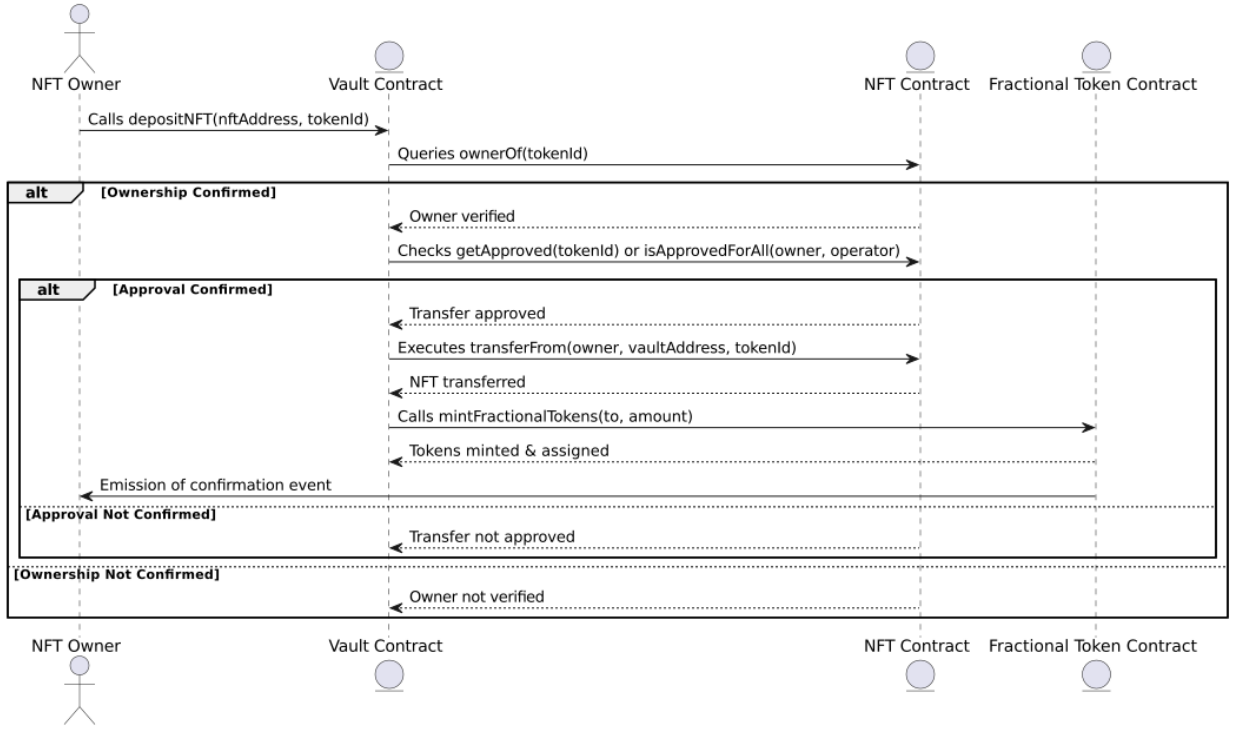


Figure 5.1: Sequence diagram illustrating the process of depositing an NFT and minting fractional tokens.

Interaction Flow:

1. **NFT Deposit Invocation:** The interaction starts with the NFT Owner executing the `depositNFT(nftAddress, tokenId)` function on the Vault Contract, initiating the deposit se-

quence. This action creates a transaction on the Ethereum blockchain, denoting the start of the NFT deposit process.

2. **Ownership and Approval Verification:** The Vault Contract proceeds to validate the NFT ownership by querying the NFT Contract with the `ownerOf(tokenId)` function. Concurrently, it checks for transfer permissions by calling `getApproved(tokenId)` or `isApprovedForAll(owner, operator)`.

3. **Conditional NFT Transfer:** Based on the outcome of the ownership and approval checks:

   - If confirmed, the Vault Contract initiates the NFT transfer via `transferFrom(owner, vaultAddress, tokenId)`.
   - If either check fails, corresponding error branches are executed, emitting failure events and aborting the process.

4. **Fractional Token Minting:** Upon a successful NFT transfer, the Vault Contract interacts with the Fractional Token Contract to mint tokens by invoking `mintFractionalTokens(to, amount)`.

5. **Token Allocation and Event Emission:** The Fractional Token Contract mints the designated tokens, allocating them to the NFT Owner's address. A confirmation event is emitted, signaling the completion of the minting process and the successful allocation of fractional tokens.

6. **Postconditions and Outcomes:** The system's final state reflects the secure storage of the NFT within the Vault Contract and the distribution of fractional tokens to the NFT Owner, enabling them to engage in subsequent ecosystem activities.

5.2.3 Governance Voting Process

The sequence diagram in Figure 5.2 shows the governance voting process within the NFT fractionalization platform, detailing the interactions between the Token Holder and the Governance Contract. The diagram showcases the comprehensive steps from proposal initiation to the proposal's outcome.

Interaction Flow:

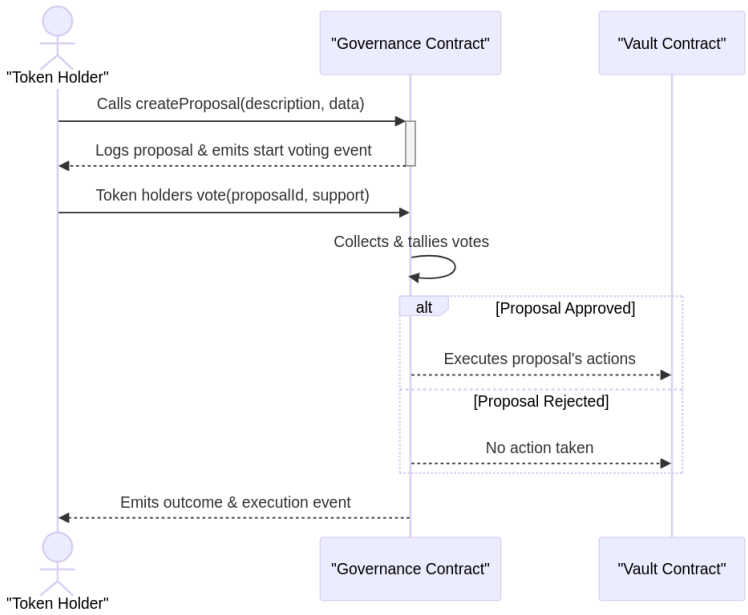


Figure 5.2: Sequence diagram depicting the governance voting process.

1. **Proposal Initiation:** The process commences with a Token Holder calling the `createProposal` function on the Governance Contract, passing a `description` of the proposal and associated `data` payload, which may contain executable code or state-changing instructions.
2. **Logging and Event Emission:** The Governance Contract logs the new proposal, assigning it a unique identifier and marking the start of the voting period. It emits a `start voting event`, signaling to all token holders that a new governance decision is underway.
3. **Voting by Token Holders:** Subsequently, Token Holders participate in the voting process by invoking the `vote` function, providing the proposal's identifier (`proposalId`) and their vote (`support` as a boolean value).
4. **Vote Collection and Tallying:** As votes are cast, the Governance Contract aggregates and tallies the votes, considering the weight of each token holder's vote based on their token balance.
5. **Conditional Execution:** Depending on the proposal's approval or rejection:
   - If the proposal is approved as determined by a defined quorum and majority rules, the Governance Contract executes the actions described in the proposal's data payload.
   - If the proposal is rejected, the process terminates with no changes made to the state, and a rejection event is emitted.

6. **Outcome Event Emission:** Regardless of the result, an outcome event is emitted, providing transparency and a permanent record of the proposal's conclusion and actions.

5.2.4 Deployment Diagram for NFT Fractionalization Ecosystem

Figure 5.3 presents a deployment diagram illustrating the architectural configuration of the NFT fractionalization solution deployed on the Ethereum Blockchain. This diagram shows the relationships and interactions among various smart contracts and external services that constitute the system's infrastructure.

1. **Ethereum Blockchain:** The primary node that hosts the suite of interrelated smart contracts, which collectively manage the fractionalization, governance, auction, and market integration aspects of NFTs.
2. **Smart Contracts:** A series of deployed contracts, each serving distinct roles within the ecosystem:
   - The *Fractional Token Contract* adheres to the ERC-20 standard and manages the minting, transferring, and burning of fractional tokens.
   - The *Governance Contract* provides mechanisms for proposing and voting on changes within the ecosystem.
   - The *Auction Contract* oversees the auctioning of NFTs or fractional tokens, facilitating bid placement and auction resolution.
   - The *Market Integration Contract* connects the ecosystem to external DeFi platforms and marketplaces, allowing for liquidity provisioning and other DeFi activities.
   - The *Vault Contract* stores NFTs and coordinates with the Fractional Token Contract to represent ownership shares.
3. **IPFS:** Utilized for decentralized storage, IPFS [149] links to the Vault Contract to securely store NFT metadata and assets, ensuring data persistence and accessibility.
4. **External DeFi Platforms and Marketplaces:** These entities interact with the Market Integration Contract, enabling token holders to engage with the broader DeFi ecosystem and

marketplace activities.

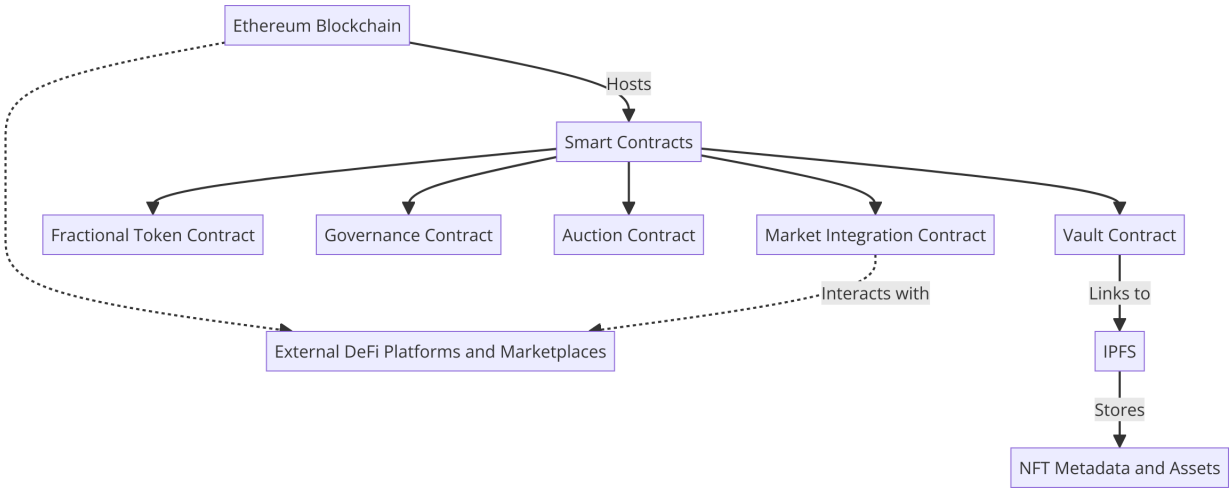


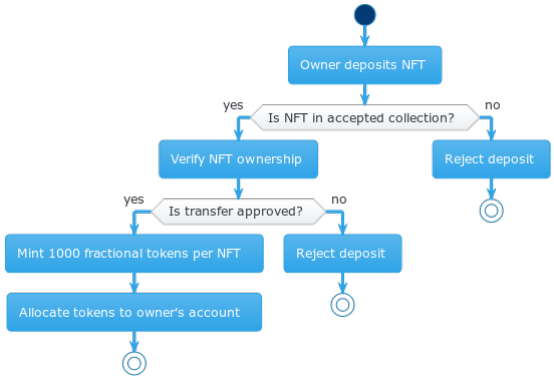
Figure 5.3: Deployment diagram depicting the smart contracts on the Ethereum Blockchain and their connection to IPFS and external DeFi platforms.

5.3 Development of the Concrete Implementation

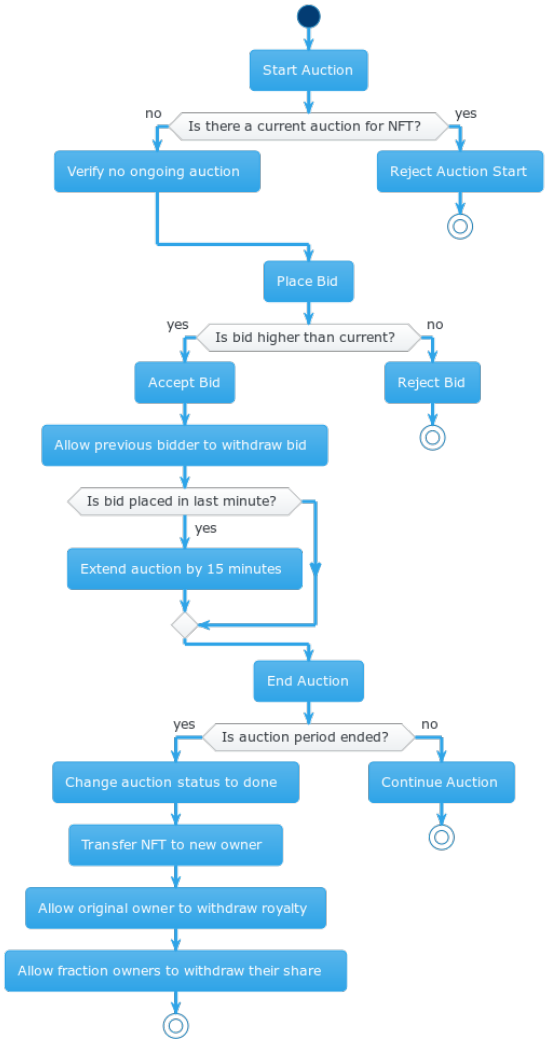
In our proposed concrete implementation of the NFT fractionalization standard [150], we focus on developing a set of smart contracts that facilitate the creation, management, and trade of fractional NFT fractions within the Ethereum ecosystem. We combined the functionalities of `IVault` and `IAuction` into a single vault contract to reduce gas costs and simplify operations. Alongside this, we introduced a governance contract for decentralized governance and a simplified, automated market contract to support fractional shares trading. We integrated fee management directly into each contract, streamlining the architecture and allowing for customized fee strategies.

5.3.1 Vault Smart Contract

In the proposed implementation of the NFT fractionalization vault, we have integrated functionalities and security measures to support the fractional ownership of NFTs, decentralized auctions, and dynamic governance. The complete code can be found in this link. The `Vault` contract's key components and their functionalities are outlined as follows:



(a) NFT management



(b) Auction management

Figure 5.4: Vault Activity diagrams

NFT Management: The NFT management process is presented in figure 5.4a. The vault facilitates the fractionalization process by enabling users to deposit and withdraw NFTs. It accepts multiple ERC721 NFTs, each belonging to the pre-specified NFT collection determined at the contract's initiation. Upon deposit, fractional tokens are minted and allocated to the depositor, correlating each NFT to 1000 fractional tokens. This mechanism simplifies the management and trading of fractions, enhancing liquidity. The exact amount of fractional tokens corresponding to an NFT (1000) must be burned for withdrawals, establishing a clear link between NFT ownership and fractional token holdings.

Auction Management: The Auction management process is presented in figure 5.4b. We incorporate a decentralized Dutch auction mechanism to enable dynamic price discovery for NFTs. Our implementation includes an `Auction` struct to manage the state of NFT auctions within the vault. It lets us track the active status, end time, highest bid, and other relevant auction details. The default duration for auctions is seven days and can be updated with governance voting. The auction is initiated via the `startAuction` function with specific parameters such as the NFT's asset address, token ID, starting price, and duration. The auction integrity is upheld by ensuring that bids exceed all previous offers and are placed before the auction concludes. Notably, if a bid is made within a predefined period before the auction's end, the duration automatically extends by 15 minutes to ensure fairness and prevent bid sniping.

Governance Functions: The vault is controlled by a governance contract, allowing parameter updates. The governance features are enhanced by enabling the governance contract address to be set post-deployment through the `setGovernanceContract` function, callable only once to prevent misuse. This governance contract can adjust auction durations, modify royalty percentages, and cancel auctions, adapting to market dynamics and community preferences. Using the `onlyGovernance` modifier ensures that only authorized calls from the governance contract can execute certain functions.

Security Considerations: Security is prioritized through various strategies, including the use of a reentrancy guard to protect against attacks, the implementation of the checks-effects-interactions pattern to prevent reentrancy issues, and access control modifiers to limit operations to authorized entities.

5.3.2 FractionalToken Smart Contract

In the proposed implementation, we introduce the `FractionalToken` contract, an ERC20 token designed to represent fractional ownership of Non-Fungible Tokens (NFTs) that extend `IFractionalToken` from the proposed NFT fractionalization standard. The complete code can be found in this link. This contract is implemented to incorporate minting and burning functionalities, essential for managing the dynamic supply of fractional tokens in accordance with the lifecycle

of NFTs within a vault.

The `FractionalToken` contract is associated with an NFT vault, indicated by the `nftVault` address variable. The NFT vault address is the sole entity authorized to mint or burn the fractional tokens. We have implemented a custom modifier, `onlyVault`, to enforce that only the designated NFT vault can execute minting and burning operations. Upon deployment, the contract's constructor initializes the token with a specified name and symbol and designates the deploying address as the contract owner.

The `mint` function facilitates the issuance of new tokens to represent fractional ownership shares of NFTs deposited in the vault. Conversely, the `burnFrom` function allows for the reduction of fractional tokens in circulation, mirroring the withdrawal or reallocation of NFT assets.

5.3.3 GovernanceContract Smart Contract

In our proposed implementation, the `GovernanceContract` facilitates governance for fraction holders, utilizing an ERC20 token representing fractional ownership as the foundation for voting. This approach ensures that decision-making power is proportionally distributed according to fractional ownership. To enhance the security and integrity of the governance process, we integrated the `TimelockController` from OpenZeppelin, introducing a mandatory delay between the approval of proposals and their execution. The complete code can be found in this link.

Figure 5.5 presents the activity diagram of the governance process. Proposals, the core of the governance mechanism, are structured to include necessary details such as a descriptive goal, the target contract address for execution, encoded function call data, and the defined voting period. Each proposal also tracks its execution status, votes for and against, total votes cast, and the total token supply at creation to determine a quorum.

In our design, initiating a proposal is conditioned upon the proposer holding a minimum threshold of the total token supply. This ensures that only stakeholders with a significant interest can propose governance actions. A quorum threshold, set at 50% of the total token supply, is required for a proposal to be considered valid.

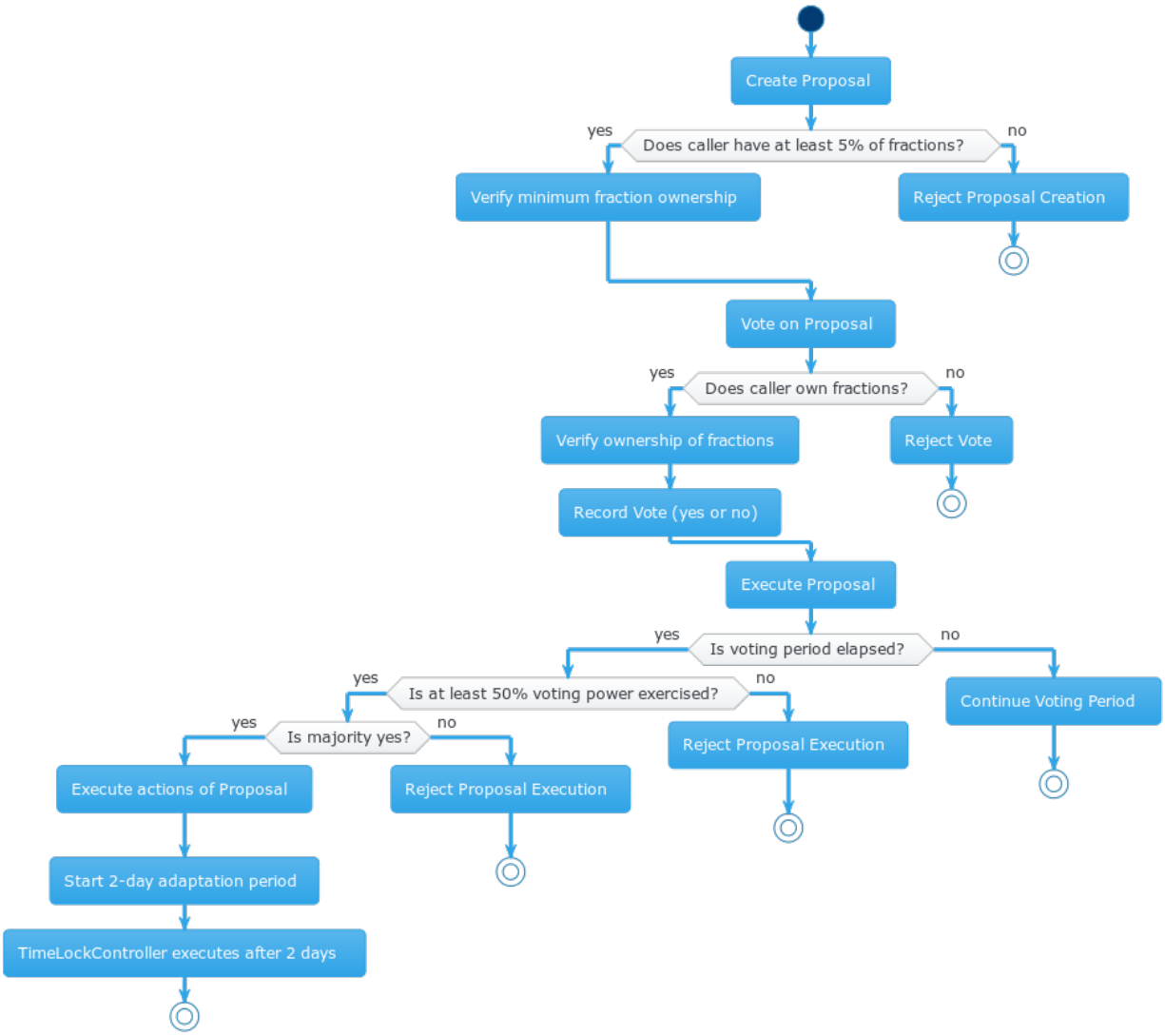


Figure 5.5: Governance mechanism

The voting process we implemented allows token holders to cast their votes on active proposals within a designated voting period, secured against reentrancy attacks to maintain the process's integrity. This design ensures that governance actions reflect the collective will of the fraction holders.

For a proposal to proceed to execution, it must conclude the voting period, achieve the necessary quorum, and receive a majority of votes in favor. Approved proposals are then scheduled for execution through the `TimelockController` after a specified delay, providing a window for review and potential intervention, enhancing the governance model's transparency and security.

5.3.4 MarketIntegration

The `MarketIntegration` contract aims to provide a simple solution for managing liquidity and enabling trades between two distinct ERC20 tokens. These tokens may represent varied assets, such as fractions of ownership in distinct NFT vaults or a mix of a fractional token and a standard ERC20 cryptocurrency like Ether (typically wrapped as WETH). This versatility underscores the contract's utility in a broad spectrum of decentralized finance (DeFi) applications; the contract illustrates the underlying mechanics of liquidity provision and token trading in a simplified manner. The complete code can be found in this link.

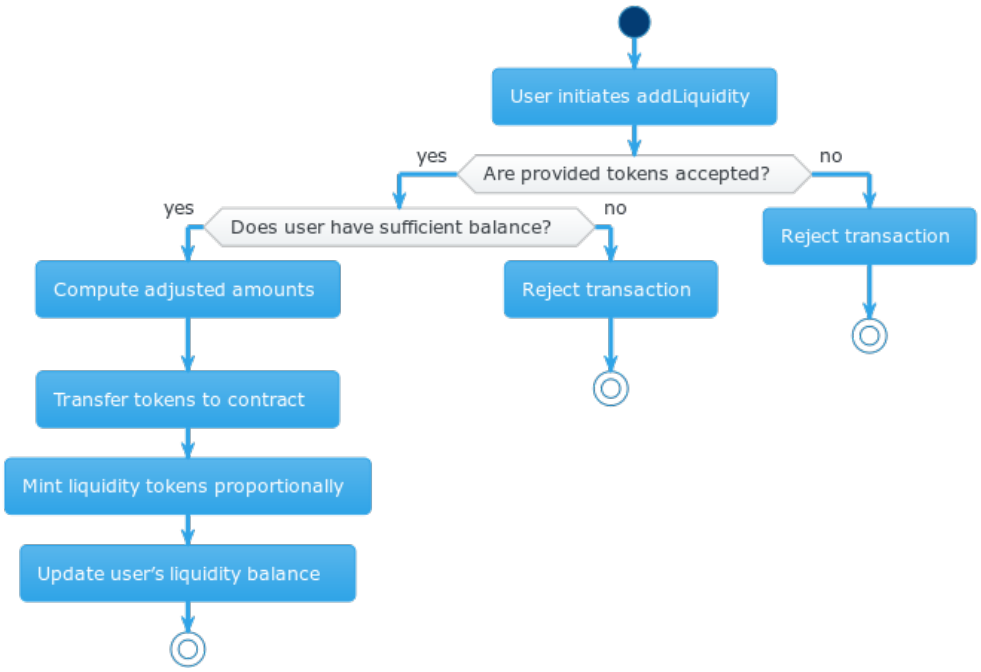


Figure 5.6: Add Liquidity

Upon initialization, the contract requires the addresses of two ERC20 tokens. These addresses enable the contract to interact with assets that could either represent ownership in various assets or a pairing of investment and utility tokens. 5.6 presents the activity diagram for adding liquidity, whereas 5.7 presents the activity diagram for removing liquidity. The addition and removal of liquidity are straightforward. Users can deposit or withdraw their tokens into or from the liquidity pool, figure

Trading between the two tokens is facilitated by a mechanism that considers the current liquidity

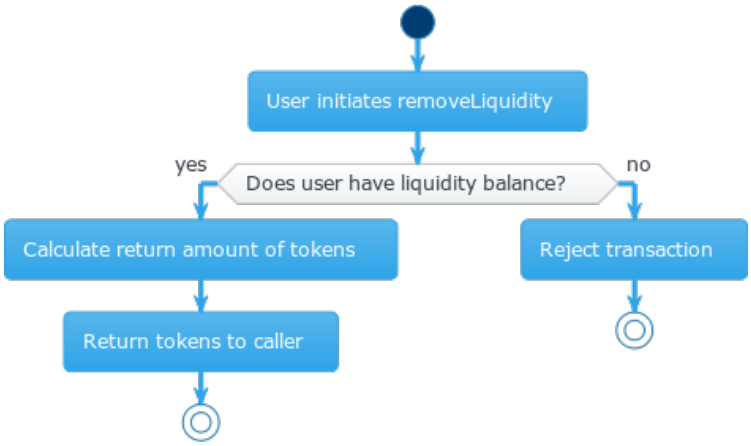


Figure 5.7: Remove Liquidity

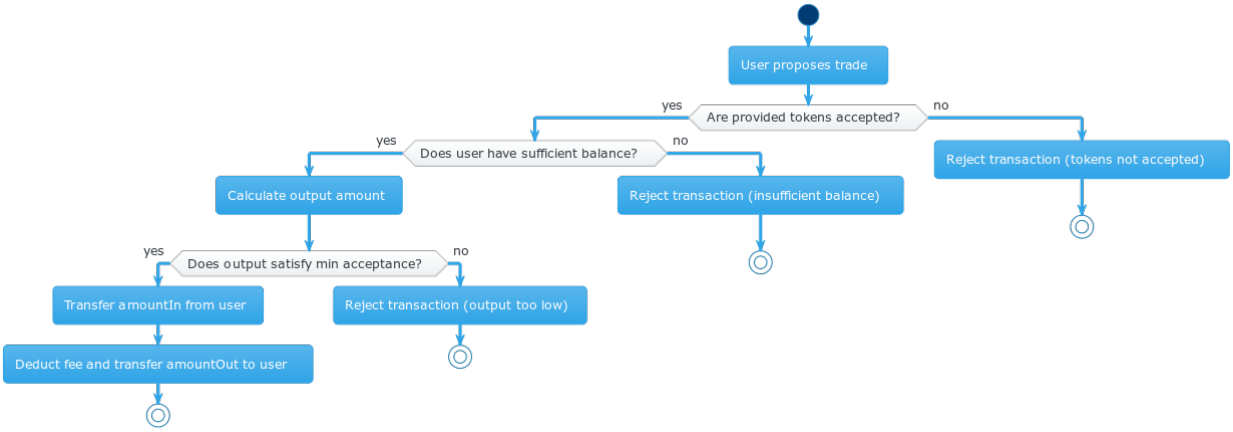


Figure 5.8: Trade Liquidity

reserves and employs slippage protection. The trading flow is presented is figure 5.8. lippage refers to the difference between the expected price of a trade and the price at which the trade is executed. It occurs due to price movement between when a transaction is initiated and when it is completed, often exacerbated in volatile market conditions. Our contract addresses slippage by allowing traders to specify a minimum acceptable amount of output tokens, minAmountOut, ensuring traders receive a value close to their expectation or the transaction fails, thus protecting them from unfavorable price movements.

The contract uses a straightforward formula to calculate the amount out for a trade, factoring in the input amount, liquidity reserves, and a predefined trading fee. Specifically, we assume a nominal fee of 0.3%, deducted to incentivize liquidity provision and mitigate potential arbitrage.

The output amount ($amountOut$) for a given input amount ($amountIn$) in a trade is calculated using the formula:

$$amountOut = \frac{amountInWithFee \times outputReserve}{(inputReserve \times 10000) + amountInWithFee}$$

Where:

- $amountInWithFee = amountIn \times fee$
- $fee = 9975$, representing a 0.3% trading fee ($10000 - fee$ gives the fee percentage).
- $inputReserve$ and $outputReserve$ are the liquidity reserves for the input and output tokens, respectively.

5.4 Deployment of the concrete Implementation

Our smart contracts are deployed within the Truffle Development Environment [148], activated through the `Truffle develop` command. This command initiates a local Ethereum blockchain simulation for development and testing, featuring instant transaction mining and a set of pre-funded accounts.

As the transaction sender for contract deployments and interactions, we used account 0, the first in the array of automatically generated accounts.

The deployment begins with the `FractionalToken` contract, labeled as "FTK," which plays a dual role within our ecosystem. As a fractional ownership token, FTK represents shares in the NFTs held within the vault, and as the governance token, grants holders the right to vote on proposals affecting the ecosystem's operations and the management of the vaulted NFTs.

Following the FTK token deployment, we deploy the `MyToken` contract. This contract is the collection accepted by the vault. Each NFT minted from `MyToken` can be deposited into the vault for fractionalization, where FTK tokens represent it.

The `Vault` contract is deployed next, configured to recognize `MyToken` as its accepted NFT

collection and utilizing FTK for the fractionalization process.

To enhance the security and integrity of the governance process, the `TimelockController` contract is then deployed. This contract introduces a mandatory delay between the approval of proposals and their execution.

The `GovernanceContract` is deployed and utilizes FTK as the voting token. This contract oversees the proposal and voting system, allowing FTK holders to influence the ecosystem's direction and decisions. Upon deployment, the GovernanceContract's address is registered with the `Vault`, integrating the governance functionalities with the vault's operations.

Finally, the `MarketIntegration` contract is deployed to enable liquidity and trading between FTK and another ERC-20 token, identified here as "TB." The entire deployment script can be found in this link.

5.5 Security Analyse

5.5.1 Slither

After compiling our smart contracts in the truffle environment, we utilized Slither to conduct a comprehensive security assessment. By running `slither .` at the project's root directory, we enabled Slither to examine all Solidity files for potential vulnerabilities. The outcome of Slither's examination revealed that four high-severity issues, two medium-severity issues, and several minor and informational errors were reported. Below, we detail each reported error alongside the solutions we opted for.

High Severity Issues:

1. Slither highlighted security risks in the `Vault.endAuction` and `Vault.redeemFractionValue` functions related to direct Ether transfers to arbitrary addresses. Such transfers can lead to reentrancy attacks or unintended code execution if the recipient is a malicious contract.

   We adopted the "Pull Over Push" strategy to mitigate these risks, significantly reducing our

contract's exposure to such threats. This approach involves:

- Creating a Pending Withdrawals Ledger: We introduced a mapping to track funds due to users, thus decoupling fund accumulation from withdrawal.
- Implementing a Secure Withdraw Function: We developed a withdraw function that lets users safely retrieve their funds, ensuring atomicity to prevent reentrancy attacks.
- Ensuring Graceful Error Handling: The withdraw function is designed to revert transactionally if Ether transfers fail, safeguarding the contract's integrity.

2. Slither's analysis identified potential vulnerabilities in the `TimelockController._execute` and `Vault.redeemFractionValue` functions due to their use of call operations for sending Ether (ETH) to arbitrary addresses.

   However, we've determined this concern to be a false positive in our context. The `Vault.redeemFractionValue` function's design, which allows users to initiate withdrawals, already mitigates the risk of reentrancy by following secure patterns. Despite Slither's warnings, our analysis concludes that the safeguards, such as the non-reentrant modifier and user-initiated withdrawal mechanisms, effectively minimize the risks associated with these operations.

3. Slither's examination of our MarketIntegration contract revealed an oversight in handling ERC20 transfer and `transferFrom` methods, where we didn't check their return values. This oversight could lead to inaccuracies in token balance accounting, posing a risk of unintended contract behaviors and vulnerabilities since these methods are expected to return a boolean indicating success or failure.

   To mitigate this, we've updated our contract to rigorously check the return values from all ERC20 token operations, employing required statements to ensure their success.

4. Slither highlighted a concern with our FractionalToken contract, specifically regarding the uninitialized state variable `isNftVaultSet`. This variable, intended to indicate if the NFT vault address has been set, was reported as never being initialized, yet it's referenced in the *updateNFTVault* function. To resolve this, we've ensured that `isNftVaultSet` is properly initialized within our contract during contract initialization.

Medium Severity Issues:

1. Slither flagged strict equality checks in OpenZeppelin's `TimelockController.getOperationState` function as potential issues, specifically for comparisons with 0 and _DONE_TIMESTAMP. These checks are vital for determining the operation's state but can pose risks if not carefully managed.

   This concern, however, is identified as a false positive. OpenZeppelin's design choice to use strict equality checks is intentional and necessary for the function's accurate operation management. While Slither's alert highlights the importance of cautious implementation, the checks are appropriate and secure in this context.

2. Slither identified potential reentrancy vulnerabilities in the `Vault.redeemFractionValue`, `MarketIntegration.removeLiquidity`, and `Vault.withdrawNFT` functions of our smart contracts. These issues stem from making external calls before updating state variables, which could allow for reentrancy attacks.

   We've implemented the Checks-Effects-Interactions pattern across our contracts to address these concerns.

Low Severity Issues and Information:

1. **Variable Shadowing in Constructor Parameters**: In the `FractionalToken`, constructor parameters `name` and `symbol` were shadowing inherited `ERC20` and `IERC20Metadata` functions. To eliminate shadowing and clarify the code, the parameters were renamed to `_name` and `_symbol`.
2. **Absence of Events for State Changes**: The `FractionalToken.updateNFTVault` function made state changes without event emission, obscuring transparency. A new event, `NFTVaultUpdated`, was introduced and emitted whenever `nftVault` is updated, enhancing contract transparency.
3. **Missing Zero Address Validation**: Functions like `Vault.setGovernanceContract` lacked validation for zero address inputs, posing a risk of logical errors. This issue was mitigated by

adding `require` statements to ensure non-zero addresses, preventing inadvertent assignment of zero addresses.

4. **External Calls Inside Loops**: In functions such as `TimelockController._execute` and `Vault.depositNFTs`, making external calls within loops raised gas and security concerns. The functions were restructured to minimize external calls within loops, ensuring gas efficiency and security.

5. **Reentrancy Vulnerabilities**: Functions including `MarketIntegration.addLiquidity` were identified as vulnerable to reentrancy attacks due to state updates after external calls. Adopting the `nonReentrant` modifier and restructuring functions to ensure state updates precede external interactions mitigated these reentrancy risks.

6. **Reliance on `block.timestamp`**: Several functions relied on `block.timestamp` for critical comparisons, which miners can slightly manipulate. The contract logic was reviewed to ensure it accounts for minor variations and potential manipulations, mitigating this issue.

7. **Usage of Low-Level Calls**: The use of low-level calls in functions like `TimelockController._execute` posed security risks. However, within OpenZeppelin's `TimelockController`, this is a necessary and deliberate choice to provide flexibility in executing arbitrary transactions. These instances are carefully implemented to manage potential reentrancy risks and ensure proper error handling.

8. **Similar Variable Names**: Variables with similar names across `MarketIntegration` and interfaces could cause confusion. By renaming variables for clarity and adopting distinct naming, this issue was mitigated to prevent confusion and improve code readability.

9. **Non-Constant State Variables**: Some state variables could have been declared as `constant` or `immutable` for gas optimization. This was addressed by reviewing and marking applicable variables as `immutable` or `constant`, optimizing gas usage and contract efficiency.

10. **Solidity Version Alignment**: Concerns were raised about the use of the latest Solidity version `^0.8.20` being potentially too recent. However, adopting the latest Solidity version by OpenZeppelin contracts aims to leverage improvements and security enhancements introduced in newer compiler versions.

11. **Inline Assembly Usage**: The presence of inline assembly in `Address.sol` was flagged for potential security risks. Inline assembly within OpenZeppelin's utilities, such as `Address.sol`, is judiciously used to perform operations not feasible with Solidity's high-level language alone. These usages are critical for certain functionalities, such as reliably detecting contract addresses, and are implemented with utmost care to maintain security.

5.5.2 Mythril

After conducting a comprehensive security analysis using Mythril on the four smart contracts, the results indicated no vulnerabilities or issues were detected. This outcome underscores the effectiveness of the coding practices adopted and the fixes implemented after the Slither analysis.

5.5.3 Echidna

We employed the Echidna testing framework to conduct an in-depth examination of the properties of our proposed concrete implementation. Below is a summary of the tests carried out; all were successfully passed.

- **FractionalToken Contract Testing:** Echidna tests for the `FractionalToken` contract focus on essential operations such as minting, burning, and supply management; the complete test can be found in this link:

  1. **Minting Authorization Test** (`echidna_test_mint`): Assesses access controls for the minting function, ensuring only authorized entities can mint tokens, thereby preventing unauthorized supply inflation.
  2. **Burning Authorization Test** (`echidna_test_burn`): Evaluates the burning function's access restrictions, confirming that only specified roles can execute burns to maintain supply integrity.
  3. **Total Supply Invariance Test** (`echidna_test_total_supply_constant`): Tests the contract's capability to maintain a constant total supply in the absence of minting or burning activities.

- **Vault Contract Testing:** Testing of the `Vault` contract within the Echidna suite evaluates governance controls, auction mechanics, and asset management, the complete test can be found in this link:

  - **Governance Contract Singleton Test** (`echidna_test_set_governance_contract_once`): Ensures the governance contract address can be set only once, cementing governance stability.
  - **Positive Auction Duration Test** (`echidna_test_auction_duration_positive`): Verifies that auction durations are set above zero, ensuring auctions occur over a meaningful timeframe.
  - **Royalty Percentage Range Test** (`echidna_test_royalty_percentage_range`): Confirms that the royalty percentage falls within a viable range (0-100%), maintaining fairness and viability in royalty mechanisms.
  - **Withdrawal Balance Check** (`echidna_test_withdraw_without_balance`): Validates that withdrawals cannot proceed without sufficient balance, safeguarding against unauthorized fund extractions.
  - **Correct Original Owner on Deposit Test** (`echidna_test_original_owner_set_on_deposit`): Ensures the original owner is correctly recorded upon NFT deposit, crucial for ownership integrity and royalty distributions.

- **Governance Mechanism Integrity:** Focuses on the `GovernanceContract`'s functionality, specifically proposal creation, voting mechanisms, and execution of governance actions, the complete test can be found in this link:

  - **Voting Increases Votes Test** (`echidna_test_voting_increases_votes`): Tests the fundamental aspect of voting, ensuring that casting a vote correctly increases the proposal's vote count, validating the vote recording mechanism.
  - **Quorum Not Met Test** (`echidna_test_quorum_not_met`): Ensures proposals without sufficient voter turnout are not incorrectly approved, maintaining the requirement for a minimum level of community engagement for governance decisions.
  - **Create Proposal Functionality Test** (`echidna_test_create_proposal`): Verifies the proposal creation process, ensuring that the system can dynamically add new proposals under appropriate conditions, keeping the governance system responsive and inclusive.

- **MarketIntegration Contract Examination:** The `MarketIntegration` contract's Echidna tests validate liquidity management, trade functionality, and supply conservation. The complete test can be found in this link:

  - **Liquidity Maintenance Test** (`echidn_test_liquidity_maintenance`): Evaluates the contract's ability to correctly manage liquidity levels, ensuring that liquidity pools are not unintentionally depleted. This test simulates various liquidity scenarios to affirm the contract's resilience and capacity to sustain market operations.
  - **Trade Execution Verification Test** (`echidna_test_trade_execution`): Assesses the contract's trade execution logic, ensuring that trades are processed only if they meet specific conditions, such as slippage tolerance and minimum liquidity requirements. This safeguards the market from potential manipulations and ensures the integrity of trade transactions.
  - **Supply Management Accuracy Test** (`echidna_test_supply_management`): Confirms the contract's effectiveness in managing token supply accurately during liquidity events. It tests the contract's response to liquidity addition and removal, verifying that supply changes reflect actual token dynamics without unintended discrepancies.

5.6 Conclusion

In this chapter, we've conducted a comprehensive technical security analysis and comparative audit of smart contracts for Tessera, NFTX, and Unicly. Utilizing tools like Slither and Mythril revealed no errors, highlighting the robustness of these platforms' contracts. Our comparative analysis of the audit errors pinpointed access control and validation errors as common vulnerabilities, underscoring the importance of focusing on these areas to enhance security. Moreover, we presented a concrete implementation of a fractionalization standard as a practical example, showcasing not only its applicability but also reinforcing the importance of rigorous security testing in validating the robustness of such implementations.

Chapter 6:

Concluding Remarks

In this thesis, we explored the concept of NFT fractionalization from both theoretical and practical viewpoints, proposing an abstract framework for smart contracts in this area. By clarifying the roles, interactions, and technical requirements involved in fractionalization, we aimed to foster a more inclusive digital asset ecosystem.

We conducted a comprehensive study that combined a review of existing literature with experimental observations to offer developers guidance on how to create secure smart contracts. Our research identified frequent vulnerabilities and recommended strategies to mitigate them while also assessing their implications for the NFT and fractionalization markets. Additionally, we evaluated the practical effectiveness of popular tools used by the community on selected smart contracts.

A crucial part of our research involved conducting a detailed security analysis and audit of smart contracts from platforms such as Tessera, NFTX, and Unicly. Tools like Slither and Mythril revealed no significant errors, attesting to the solidity of these platforms' contracts. Our audit pinpointed common vulnerabilities, particularly in access control and validation, underscoring the need for focused improvements. Additionally, we illustrated the viability of our standardization proposal with a concrete implementation, emphasizing the importance of rigorous security testing to affirm the strength of such frameworks.

Looking forward, we propose to further this work by advocating for our standard within the Ethereum community. This will involve initiating discussions on Ethereum community forums and drafting a specification for submission as an Ethereum Improvement Proposal (EIP). This effort

aims to enhance blockchain infrastructure and governance, promoting a move towards greater standardization and security. For future work, we aim to delve deeper into analyzing gas consumption and associated fees for each operation within smart contracts. Additionally, we plan to refine our proposed solution by incorporating considerations of gas usage to enhance its effectiveness.

In summary, this thesis contributes to the academic understanding of NFT fractionalization and offers practical insights and tools for its safe and efficient implementation. It lays the groundwork for future research and development in blockchain technology, aiming to improve the security, liquidity, and inclusivity of the NFT market.

Bibliography

[1] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges, 2021.

[2] Akash Takyar. How to create a multifunctional fractionalized nft? https://www.leewayhertz.com/fractionalized-nft/, 2022. Accessed: 2023-02-01.

[3] BV Buterin. Ethereum white papee. *https://ethereum.org/en/whitepaper/*, 2014.

[4] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, may 2019.

[5] Haouari Wejdene and Fokaefs Marios. Towards a standardization of fractionalization smart contracts for non-fungible tokens. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, CASCON '23, page 132–141, USA, 2023. IBM Corp.

[6] Wejdene Haouari, Abdelhakim Senhaji Hafid, and Marios Fokaefs. Vulnerabilities of smart contracts and mitigation schemes: A comprehensive survey, 2024.

[7] Mohammadreza Rasolroveicy, Wejdene Haouari, and Marios Fokaefs. Public or private? a techno-economic analysis of blockchain. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, CASCON '21, page 83–92, USA, 2021. IBM Corp.

[8] M.N.M. Bhutta, A.A. Khwaja, A. Nadeem, H.F. Ahmad, M.K. Khan, M.A. Hanif, Houbing Song, M. Alshamari, and Yue Cao. A survey on blockchain technology: Evolution, architecture and security. *IEEE Access*, 9:61048 – 73, 2021//.

[9] Xiang Fu, Huaimin Wang, and Peichang Shi. A survey of blockchain consensus algorithms: mechanism, design and applications. *Science China Information Sciences*, 64(2):121101 (15 pp.) –, 2021/02/.

[10] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: 2023-08-22.

[11] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and M. Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475 – 91, 2020/04/.

[12] William Mougayar. *The Business Blockchain: Promise, Practice, and Application of the Next Internet Technology*. John Wiley & Sons, 2016.

[13] Solidity Team. Solidity official website . https://soliditylang.org/, 2022. Accessed: 2022-10-02.

[14] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[15] Martin Becze and Hudson Jameson. Ethereum improvement proposals. Ethereum Improvement Proposals, 2015.

[16] Fabian Vogelsteller and Vitalik Buterin. Erc-20: Token standard. Ethereum Improvement Proposals, 2017.

[17] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Erc-721: Non-fungible token standard. Ethereum Improvement Proposals, 2017.

[18] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. Erc-1155: Multi token standard. Ethereum Improvement Proposals, 2018.

[19] Christian Reitwießner, Nick Johnson, Fabian Vogelsteller, Jordi Baylina, Konrad Feldmeier, and William Entriken. Erc-165: Standard interface detection. Ethereum Improvement Proposals, 2018.

[20] Vijay Mohan. Automated market makers and decentralized exchanges: a defi primer. *Financial Innovation*, 8(1):20, 2022.

[21] uniswap. Uniswap protocol. https://uniswap.org/, 2023. Accessed: 2023-11-23.

[22] Lucius Fang. Fractionalized nft. CoinGecko, 2021.

[23] Adarsh Vijayakumaran. Democratizing nfts: F-nfts, daos and securities law. *Richmond Journal of Law and Technology*, 2021, November 2021. Available at SSRN: https://ssrn.com/abstract=3964905 or http://dx.doi.org/10.2139/ssrn.3964905.

[24] Wonseok Choi, Jongsoo Woo, and James Hong. Fractional non-fungible tokens: Overview, evaluation, marketplaces, and challenges. *International Journal of Network Management*, 01 2024.

[25] Nataliia Bogdanova1 Yehor Rudytsia1. Uml model of the property right distribution module using nft fractionalization based on blockchain technology. *International Science Journal of Engineering and agriculture*, 1(3):98–109, 2022.

[26] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019.

[27] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. *CoRR*, abs/1910.10601, 2019.

[28] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022.

[29] Dominik Harz and William J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *CoRR*, abs/1809.09805, 2018.

[30] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10:6605–6621, 2022.

[31] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[32] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3), 2020.

[33] Lie-Huang Zhu, Bao-Kun Zheng, Meng Shen, Feng Gao, Hong-Yu Li, and Ke-Xin Shi. Data security and privacy in bitcoin system: A survey. *Journal of Computer Science and Technology*, 35(4):843–862, jul 2018.

[34] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853, 2020.

[35] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, Daehun Nyang, and David Mohaisen. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, PP:1–1, 03 2020.

[36] Edward Wilson. What are fractionalized nfts?s. `https://www.argent.xyz/learn/fractionalized-nfts/#:~:text=are%20fractionalized%20NFTs%3F-`, 2021. Accessed: 2023-02-01.

[37] Luke O'Neill. The rise and risks of fractionalized nfts. https://blockpit.io/unchained/en/fractionalized-nfts/, 2022. Accessed: 2023-05-02.

[38] KPMG. Navigating the adoption of nfts. https://kpmg.com/ca/en/home/insights/2022/08/navigating-the-adoption-of-nfts.html, 2022. Accessed: 2023-02-01.

[39] openzeppelin. Erc 721. https://docs.openzeppelin.com/contracts/2.x/api/token/erc721, 2023. Accessed: 2023-02-01.

[40] openzeppelin. Erc 1155. https://docs.openzeppelin.com/contracts/3.x/api/token/erc1155#IERC1155Receiver, 2023. Accessed: 2023-02-01.

[41] Arnav Vohra. Exploring the buyout mechanism in nft fractionalization. https://medium.com/nibbl/exploring-the-buyout-mechanism-in-nft-fractionalization-ab6722b50d19, 2021. Accessed: 2023-02-01.

[42] Will Hunter. Mechanism series: Last price dutch auction. https://medium.com/tessera-nft/mechanism-series-last-price-dutch-auction-f65861c8a619, 2022. Accessed: 2023-05-02.

[43] Brandon Galang. Getting started with tessera. https://medium.com/tessera-nft/getting-started-with-tessera-c9709293ee88, 2022. Accessed: 2023-02-01.

[44] jacob. Hyperstructures. https://jacob.energy/hyperstructures.html, 2022. Accessed: 2024-01-28.

[45] smaroo Shant andy8052. Fractional contracts. https://github.com/code-423n4/2022-07-fractional. Accessed: 2023-10-23.

[46] ZeframLou wighawag. Cloneswithimmutableargs contract. https://github.com/wighawag/clones-with-immutable-args/tree/master. Accessed: 2023-10-23.

[47] openzeppelin. Merkleproof openzeppelin. https://docs.openzeppelin.com/contracts/3.x/api/cryptography#MerkleProof. Accessed: 2023-11-23.

[48] Jacob Evans Remco Bloemen, Leonid Logvinov (@LogvinovLeon). Typed structured data hashing and signing. Ethereum Improvement Proposals, 2017.

[49] unicly. unic.ly. https://www.app.unic.ly/#/, 2023. Accessed: 2023-02-10.

[50] Vijay Mohan. Automated market makers and decentralized exchanges: a defi primer. *Financial Innovation*, 8(1):20, 2022.

[51] Unicly. Unicly litepaper. https://docs.unic.ly/, 2022. Accessed: 2023-02-01.

[52] Geoffrey Hayes Robert Leshner. Compound: The money market protocol. https://compound.finance/documents/Compound.Whitepaper.pdf, 2018.

[53] Sushimaker. https://etherscan.io/address/0xe11fc0b43ab98eb91e9836129d1ee7c3bc95df50, 2023. Accessed: 2023-10-31.

[54] randao. Governoralpha.sol smart contract. https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/GovernorAlpha.sol. Accessed: 2023-11-20.

[55] NFTX. Introduction to nftx. https://docs.nftx.io/, 2022. Accessed: 2023-05-01.

[56] fractional art. Fractional v2 contest findings & analysis report. https://code4rena.com/reports/2022-07-fractional/, 2022. Accessed: 2023-02-01.

[57] Quantstamp. Unicly art audit report. https://docs.unic.ly/smart-contracts/security-audit, 2020. Accessed: 2023-02-01.

[58] SECBIT. Secbit audit. https://docs.nftx.io/v/main/smart-contracts/bug-bounty/secbit-audit, 2022. Accessed: 2024-01-02.

[59] Trail of Bits. Nftx protocol v2 security assessment. https://docs.nftx.io/v/main/smart-contracts/bug-bounty/trail-of-bits-audit, 2022. Accessed: 2024-01-02.

[60] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, New York, NY, USA, 2016. Association for Computing Machinery.

[61] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering – a systematic literature review. *Information and Software Technology*, 51(1):7–15, 2009.

[62] anonymous authors. Dforce network - rekt. https://rekt.news/dforce-network-rekt/. Accessed: 2023-10-05.

[63] Thomas Claburn. Thief milks cream finance for $18m+ in cryptocurrency after spotting security bug. https://www.theregister.com/2021/08/31/cream_finance_theft/. Accessed: 2022-04-24.

[64] quadrigainitiative. Description of events. https://www.quadrigainitiative.com/casestudy/sirenmarketreentrancybug.php. Accessed: 2022-04-24.

[65] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018.

[66] openzeppelin. openzeppelin security. https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard. Accessed: 2022-04-20.

[67] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927, 2020.

[68] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.

[69] nist. Cve-2018-10299 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-10299. Accessed: 2022-04-20.

[70] Weili Chen, Zibin Zheng, Edith C.-H. Ngai, Peilin Zheng, and Yuren Zhou. Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access*, 7:37575–37586, 2019.

[71] kingoftheether. Post-mortem investigation (feb 2016). https://www.kingoftheether.com/postmortem.html. Accessed: 2022-04-20.

[72] openzeppelin. openzeppelin the parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. Accessed: 2022-04-20.

[73] Xiaolong Liu, Riqing Chen, Yu-Wen Chen, and Shyan-Ming Yuan. Off-chain data fetching architecture for ethereum smart contract. In *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*, pages 1–4, 2018.

[74] Dan Boneh and Moni Naor. Timed commitments. In *Annual International Cryptology Conference*, CRYPTO '00, page 236–254, Berlin, Heidelberg, 2000. Springer-Verlag.

[75] randao. randao github. https://github.com/randao/randao. Accessed: 2022-04-20.

[76] Chainlink. Chainlink vrf. https://docs.chain.link/vrf, 2023. Accessed: 2024-01-02.

[77] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 22–29, 2020.

[78] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities. In *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 107–111, 2020.

[79] Bo Jiang, Ye Liu, and W.K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, 2018.

[80] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium*, 01 2018.

[81] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *40th International Conference on Software Engineering*, ICSE '18, page 65–68, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Antonio López Vivar, Ana Lucila Sandoval Orozco, and Luis Javier García Villalba. A security framework for ethereum smart contracts. *Computer Communications*, 172:119–129, 2021.

[83] Ardit Dika and Mariusz Nowostawski. Security vulnerabilities in ethereum smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 955–962, 2018.

[84] Tomas Krupa, Michal Ries, Ivan Kotuliak, Kristi'an Košťál, and Rastislav Bencel. Security issues of smart contracts in ethereum platforms. In *2021 28th Conference of Open Innovations Association (FRUCT)*, pages 208–214, 2021.

[85] Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *20th International Conference on Information Integration and Web-based Applications & Services*, iiWAS2018, page 375–380, New York, NY, USA, 2018. Association for Computing Machinery.

[86] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC*, New York, NY, USA, 2016. Association for Computing Machinery.

[87] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode, 2021.

[88] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Maximilian Wöhrer and Uwe Zdun. Design patterns for smart contracts in the ethereum ecosystem. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1513–1520, 2018.

[90] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *34th Annual Computer Security Applications Conference*, ACSAC '18, page 664–676, New York, NY, USA, 2018. Association for Computing Machinery.

[91] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[92] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 454–469, New York, NY, USA, 2020. Association for Computing Machinery.

[93] swcregistry. Smart contract weakness classification and test cases. https://swcregistry.io/. Accessed: 2022-04-02.

[94] securing. Smart contract security verification standard. https://github.com/securing/SCSVS. Accessed: 2022-04-02.

[95] Wolfgang Wögerer and Technische Universität Wien. A survey of static program analysis techniques, 2005.

[96] owasp. Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis. Accessed: 2022-04-02.

[97] Haijun Wang, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. Oracle-supported dynamic exploit generation for smart contracts, 2019.

[98] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *CPP 2018: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 66–77, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020.

[100] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020.

[101] ethereum. Act formal specification. https://ethereum.github.io/act/. Accessed: 2022-04-20.

[102] consensys. Blockchain security & ethereum smart contract audits. https://consensys.net/diligence/, 2022. Accessed: 2022-05-12.

[103] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[104] Lukasz Mazurek. *EthVer: Formal Verification of Randomized Ethereum Smart Contracts*, pages 364–380. Springer-Verlag, 09 2021.

[105] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC '18: Proceedings of the 34th Annual Computer Security Applications Conference*, New York, NY, USA, 2018. Association for Computing Machinery.

[106] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560, New York, NY, USA, 2020. Association for Computing Machinery.

[107] Joshua Ellul and Gordon J. Pace. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 158–163, 2018.

[108] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1398–1409, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Andrew William Roscoe. Reguard: Finding reentrancy bugs in smart contracts. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

[110] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering*, pages 1186–1189, 11 2019.

[111] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*, pages 243–269. Springer International Publishing, 04 2018.

[112] crytic. slither tool github. https://github.com/crytic/slither. Accessed: 2023-05-31.

[113] CONSENSYS. Mythx officil website. https://mythx.io/. Accessed: 2022-03-28.

[114] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. https://conference.hitb.org/hitbsecconf2018ams/materials/WHITEPAPERS/WHITEPAPER%20-%20Bernhard%20Mueller%20-%20Smashing%20Ethereum%20Smart%20Contracts%20for%20Fun%20and%20ACTUAL%20Profit.pdf, 2018.

[115] ConsenSys. mythril tool github. https://github.com/ConsenSys/mythril. Accessed: 2023-05-31.

[116] crytic. Echidna tool github. https://github.com/crytic/echidna. Accessed: 2023-05-31.

[117] trailofbits. manticore tool github. https://github.com/trailofbits/manticore. Accessed: 2023-05-31.

[118] eth sri. securify2 tool github. https://github.com/eth-sri/securify2. Accessed: 2023-05-31.

[119] runtimeverification. Kevm tool github. https://github.com/runtimeverification/evm-semantics. Accessed: 2023-05-31.

[120] smartdec. smartcheck tool github. https://github.com/smartdec/smartcheck. Accessed: 2023-05-31.

[121] nevillegrech. Madmax tool github. https://github.com/nevillegrech/MadMax. Accessed: 2023-05-31.

[122] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. Practical mutation testing for smart contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 289–303. Springer International Publishing, 2019.

[123] JoranHonig. vertigo tool github. https://github.com/JoranHonig/vertigo. Accessed: 2023-05-31.

[124] SeUniVr. Ethersolve tool github. https://github.com/SeUniVr/EtherSolve. Accessed: 2022-03-28.

[125] pventuzelo. octopus tool github. https://github.com/pventuzelo/octopus. Accessed: 2023-05-31.

[126] enzymefinance. Oyente tool github. https://github.com/enzymefinance/oyente. Accessed: 2023-05-23.

[127] openzeppelin. erc20-verifier. https://erc20-verifier.openzeppelin.com/. Accessed: 2022-04-11.

[128] raineorshine. solgraph tool github. https://github.com/raineorshine/solgraph. Accessed: 2023-05-31.

[129] christoftorres. Osiris tool github. https://github.com/christoftorres/Osiris. Accessed: 2023-05-23.

[130] Programming z3. http://theory.stanford.edu/~nikolaj/programmingz3.html, 2022. Accessed: 2022-04-17.

[131] SmartContractSecurity. Swc-101 test case. https://swcregistry.io/docs/SWC-101#integer-overflow-mapping-sym-1sol. Accessed: 2022-04-02.

[132] Purathani Praitheeshan, Lei Pan, and Robin Doss. *Security Evaluation of Smart Contract-Based On-chain Ethereum Wallets*, pages 22–41. Springer-Verlag, 12 2020.

[133] crytic. slither list of vulnerabilities. https://github.com/crytic/slither#detectors. Accessed: 2022-04-02.

[134] crytic. Slither erc conformance. https://github.com/crytic/slither/wiki/ERC-Conformance. Accessed: 2022-04-20.

[135] crytic. Crytic website. https://www.crytic.io/. Accessed: 2022-03-30.

[136] ConsenSys. mythril modules. https://mythril-classic.readthedocs.io/en/master/module-list.html. Accessed: 2022-06-11.

[137] ConsenSys. Mythx and continuous integration (part 1): Circleci. https://blog.mythx.io/howto/mythx-and-continuous-integration-part-1-circleci/. Accessed: 2022-05-11.

[138] trailofbits. Category archives: Manticore. https://blog.trailofbits.com/category/manticore/. Accessed: 2022-04-22.

[139] trailofbits. List of ethereum detectors. https://github.com/trailofbits/manticore/wiki/Ethereum-Detectors. Accessed: 2022-06-11.

[140] Quickcheck: Automatic testing of haskell programs. https://hackage.haskell.org/package/QuickCheck. Accessed: 2023-05-31.

[141] trailofbits. Echidna, a smart fuzzer for ethereum. https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/. Accessed: 2023-05-31.

[142] crytic. Testing a property with echidna. https://github.com/crytic/building-secure-contracts/blob/master/program-analysis/echidna/introduction/how-to-test-a-property.md. Accessed: 2023-05-31.

[143] code4rena. Secure your smart contracts. https://code4rena.com/, 2023. Accessed: 2023-02-01.

[144] Quantstamp. Securing the future of web3. https://quantstamp.com/, 2020. Accessed: 2023-02-01.

[145] Trail of Bits. We don't just fix bugs, we fix software. https://www.trailofbits.com/, 2023. Accessed: 2024-01-02.

[146] SECBIT. Secbit. https://secbit.io/, 2023. Accessed: 2024-01-02.

[147] owasp. Owasp risk rating methodology. https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. Accessed: 2023-08-02.

[148] ConsenSys. Truflle suite. https://archive.trufflesuite.com//. Accessed: 2023-12-05.

[149] IPFS. Best practices for storing nft data using ipfs. https://docs.ipfs.tech/how-to/best-practices-for-nft-data/#types-of-ipfs-links-and-when-to-use-them, 2023. Accessed: 2024-01-11.

[150] wejdeneHaouari. Nft fractionalization implementation github. hhttps://github.com/wejdeneHaouari/NFT-F. Accessed: 21-03-2024.

Appendix A

Tessera Smart Contracts

Attribute	Description	Type/Data
implementation	Stores the address of the Vault proxy contract for cloning.	address
nextSeeds	Maps an account address to its next seed value for clone deployment.	mapping(address => bytes32)

Table A.1: VaultFactory Attributes

Function	Description	Modifiers/Conditions
constructor	Initializes the ‘implementation‘ with a new ‘Vault‘ contract address.	public
deploy	Deploy a new vault for the message sender.	external
deployFor	Deploys a vault for a specified owner, with front-running prevention.	public
getNextAddress	Computes the next vault's address for a given account.	external, view
getNextSeed	Retrieves the next seed for a given deployer.	external, view

Table A.2: VaultFactory Functions

Attribute	Description	Type/Data
owner	The address of the vault owner, who has control over the contract.	address
merkleRoot	Merkle root hash for vault permissions.	bytes32
nonce	Used as an initializer value to prevent re-initialization of the contract.	uint256
MIN_GAS_RESERVE	A constant representing the minimum gas reserve for operations within the contract.	uint256
methods	Maps function selectors to plugin addresses, enabling the dynamic extension of contract capabilities.	mapping(bytes4 => address)

Table A.3: Vault Attributes

Function	Description	Modifiers/Conditions
init	Initializes the contract by setting the owner and nonce.	external
execute	Allows execution of transactions through delegatecall, with permission verification via Merkle proofs.	external, payable
install	install plugins by mapping function selectors to contract addresses.	external
setMerkleRoot	Updates the Merkle root hash used for permission verification.	external
transferOwnership	Transfers contract ownership to a new address.	external
uninstall	Remove plugin mappings, effectively uninstalling the plugin.	external
_execute	Internal function to perform delegatecalls to target addresses.	internal
_revertedWithReason	Extracts and reverts with the error reason from failed delegatecalls.	internal, pure

Table A.4: Vault Functions

Attribute	Description	Type/Data
factory	Address of the VaultFactory contract used to deploy new vaults.	address
fNFT	Address of the FERC1155 token contract instance used for minting and burning tokens.	address
fNFTImplementation	Address of the FERC1155 implementation used for creating fNFT contract clones.	address
nextId	Maps a collection address to its next available token ID, facilitating token management.	mapping(address => uint256)
vaultToToken	Maps a vault address to its corresponding token information, including the fNFT address and token ID.	mapping(address => VaultInfo)

Table A.5: VaultRegistry Attributes

Function	Description	Modifiers/Conditions
create	Deploys a new vault with specified permissions and plugins.	external
createFor	Deploys a new vault for a given owner with permissions and plugins.	external
mint	Mints fNFT tokens associated with a vault to a specified address.	external
burn	Burns fNFT tokens associated with a vault from a specified address.	external
totalSupply	Returns the total supply of fNFT tokens for a given vault.	external, view
uri	Retrieves the URI for fNFT tokens associated with a vault.	external, view
createCollection	Creates a new vault and corresponding fNFT collection for the message sender.	external
createInCollection	Creates a new vault within an existing fNFT collection.	external

Table A.6: VaultRegistry Functions

Attribute	Description	Type
NAME	The name of the token contract.	string constant
VERSION	The version number of the token contract.	string constant
_controller	Address that can deploy new vaults and manage metadata.	address
contractURI	URI for the contract metadata.	string
isApproved	Tracks approvals for token types.	mapping
metadata	Mapping of token ID types to metadata addresses.	mapping
nonces	Tracks account nonces for metadata transactions.	mapping
totalSupply	Tracks total supply for token ID types.	mapping
royaltyAddress	Tracks royalty receivers for token ID types.	mapping
royaltyPercent	Tracks the royalty percent for token ID types.	mapping

Table A.7: Attributes of FERC1155

Function	Description	Modifiers
burn	Burns a specified amount of tokens for a given ID.	onlyRegistry
mint	Mints new tokens for a given ID.	onlyRegistry
setApprovalFor	Sets scoped approval for a single token ID.	
permit	Approves an operator for a token type with a valid signature.	
permitAll	Approves an operator for all token types with a valid signature.	
setContractURI	Sets the URI for contract metadata.	onlyController
setMetadata	Associates a token ID with a metadata contract.	onlyController
setRoyalties	Sets royalty information for a token ID.	onlyController
transferController	Transfer the controller role to a new address.	onlyController
royaltyInfo	Provides royalty information for a token ID.	
safeTransferFrom	Transfers tokens between accounts.	
uri	Returns the URI for a token's metadata.	

Table A.8: Key Functions of FERC1155

Attribute	Description	Type
supply	The address of the Supply target contract, indicating where the mint function is located.	address

Table A.9: Attributes of the Minter Contract

Function	Description	Modifiers
`getLeafNodes`	Generates and returns a list of Merkle tree leaf nodes derived from the contract's permissions.	`external view`
`getPermissions`	Provides a detailed list of permissions that govern what actions the contract can execute.	`public view`
`_mintFractions`	Core function that mints fractional tokens, leveraging permissions for secure execution.	`internal`

Table A.10: Key Functions of the Minter Contract

Attribute	Description	Type
registry	Address of the VaultRegistry contract.	address
supply	Address of the Supply target contract.	address
transfer	Address of the Transfer target contract.	address
PROPOSAL_PERIOD	Duration of the proposal period for a buyout auction.	uint256
REJECTION_PERIOD	Duration of the rejection period for a buyout auction.	uint256
buyoutInfo	Mapping of vault addresses to their respective auction details.	mapping(address => Auction)

Table A.11: Buyout Contract Attributes

Function	Description	Modifiers
start	Initiate the buyout auction for a vault.	external payable
sellFractions	Allows users to sell fractional tokens to the buyout pool for ether.	external
buyFractions	Enables users to buy fractional tokens from the buyout pool with ether.	external payable
end	Concludes the buyout auction, determining its success or failure.	external
cash	Permits users to cash out their share from a successful buyout.	external
redeem	Facilitate the redemption of underlying assets from the vault in case of an inactive buyout.	external
withdrawERC20	Withdraw an ERC-20 token from the vault.	external
withdrawERC721	withdraw an ERC-721 token from the vault.	external
withdrawERC1155	Withdraw an ERC-1155 token from the vault.	external
batchWithdrawERC1155	Batch withdraws multiple ERC-1155 tokens from the vault.	external
getLeafNodes	Retrieves the list of leaf nodes for generating a Merkle tree.	external view
getPermissions	Acquires the list of permissions for the contract.	public view

Table A.12: Buyout Contract Functions

Attribute	Description	Type
buyout	Address of the Buyout module contract	address payable
registry	Address of the VaultRegistry contract	address
nextId	Counter for assigning unique IDs	uint256
PROPOSAL_PERIOD	Duration for which a migration proposal remains active.	uint256
migrationInfo	Maps a vault address and proposal ID	mapping(address => mapping(uint256 => Proposal))
userProposalEth	Tracks ether contributions by users to a proposal ID.	mapping(uint256 => mapping(address => uint256))
userProposalFractions	Records fractional token contributions by users towards a proposal.	mapping(uint256 => mapping(address => uint256))

Table A.13: Attributes of the Migration Contract

Function	Description	Modifiers
propose	Initiate a migration proposal, setting the stage for vault evolution.	external
join	Allows users to support a migration proposal by contributing ether and fractional tokens.	external payable, nonReentrant
leave	Enables contributors to retract their support and recover their contributions.	external
commit	Commence the buyout process for a migration proposal, given target conditions are met.	external
settleVault	Finalizes the migration, creating a new vault with updated permissions.	external
settleFractions	Mints new fractional tokens for the newly created vault post-migration.	external
withdrawContribution	withdraw contributions from unsuccessful migration proposals.	external
migrateVaultERC20	migrate ERC-20 assets to the new vault as part of the migration process.	external
migrateVaultERC721	Transfer ERC-721 assets to the newly established vault following migration.	external
migrateVaultERC1155	move ERC-1155 assets to the new vault, completing the asset migration.	external
batchMigrateVaultERC1155	Facilitates the batch migration of multiple ERC-1155 assets.	external
migrateFractions	Transfers fractional ownership from the old to the new vault structure.	external

Table A.14: Key Functions of the Migration Contract

Attribute	Type	Description
registry	address	The address of the VaultRegistry contract

Table A.15: Attributes of the Supply Contract

Function	Description	Modifiers
mint	Mints fractional tokens to an address using assembly.	external
burn	Burns fractional tokens from an address utilizing assembly.	external

Table A.16: Functions of the Supply Contract

Function	Description	Modifiers
ERC20Transfer	Transfers ERC-20 tokens to a specified address. Optimized for gas efficiency using low-level calls.	external
ERC721TransferFrom	Facilitates the transfer of an ERC-721 token from one address to another. Utilizes assembly for optimized execution.	external
ERC1155TransferFrom	Enables the transfer of ERC-1155 tokens, supporting both single and batch transfers through direct contract interaction.	external

Table A.17: Key Functions of the Transfer Contract

Appendix B

Unicly Smart Contracts

Attribute	Description	Type
name	Token's official name	String
symbol	An abbreviated representation, or symbol, for the token.	String
_delegates	Maps each account to its delegate, enabling delegation of the voting power.	Mapping
checkpoints	record the vote checkpoints for each account by index, allowing tracking of historical votes.	Mapping
numCheckpoints	Maintains the number of vote checkpoints for each account.	Mapping
nonces	States required for signing and validating signatures, ensuring unique votes per signatory.	Mapping

Table B.1: Unic Attributes

Function	Description	Modifiers
mint	Mints a specified amount of tokens and assigns them to an address, increasing the total supply.	onlyOwner
delegate	Allows an account to delegate its voting rights to another account.	None
delegateBySig	Enables delegation of voting rights using a signature, facilitating delegation without direct transaction from the delegator.	None
_delegate	An internal function that handles the process of transferring voting rights from one account to another.	Internal
_moveDelegates	Internally manages the shift in delegated votes when tokens are transferred or delegated.	Internal
_writeCheckpoint	Records a vote checkpoint for an account, enabling tracking of voting power changes over time.	Internal

Table B.2: Unic Functions

Attribute	Description	Type
feeTo	Address designated to receive fees.	address
feeToSetter	Address with permission to change ‘feeTo‘.	address
uTokens	List storing addresses of all uTokens.	address[]
getUToken	Maps uToken’s address to its position in ‘uTokens‘.	Mapping

Table B.3: UnicFactory Attributes

Function	Description	Modifiers
createUToken	Generate a new uToken with given attributes.	None
setFeeTo	update the ‘feeTo‘ address.	onlyFeeToSetter
setFeeToSetter	Modifies the ‘feeToSetter‘ address.	onlyFeeToSetter
uTokensLength	Returns total number of uTokens.	None

Table B.4: UnicFactory Functions

Attribute	Description	Type/Data
currentNFTIndex	Tracks the current index and length of NFTs within the contract.	uint256
active	Indicates whether NFTs can be withdrawn. When true, withdrawals are restricted.	bool
totalBidAmount	Represents the total amount of bids placed within the contract.	uint256
unlockVotes	Tracks the total number of votes cast to unlock the collection.	uint256
_threshold	The minimum threshold of votes required to unlock the collection.	uint256
issuer	Stores the address of the issuer of the contract.	address
_description	A descriptive string about the contract or collection.	string
cap	Sets the maximum cap for the token supply.	uint256
factory	Reference to the UnicFactory contract.	IUnicFactory

Table B.5: Converter Attributes

Function	Description	Modifiers/Conditions
deposit	Allows depositing of NFTs, with restrictions on token type and quantity.	external
issue	Locks NFTs as collateral and issues tokens to the issuer. Activates the contract.	external
refund	Permits refunding of NFTs to the issuer before activation of the contract.	external
bid	Enables placing bids on NFTs, subject to certain conditions and thresholds.	external, payable
unbid	Allows withdrawal of a bid under specific circumstances.	external
claim	Facilitate the claiming of NFTs by the winning bidder post-threshold achievement.	external
approveUnlock	approve unlocking of the collection by depositing tokens.	external
unapproveUnlock	Reverse the token deposit made for unlocking the collection.	external
redeemETH	Enables redeeming ETH in proportion to the user's uToken amount after reaching the threshold.	external
getBlockTimestamp	Internal function to return the current block timestamp.	internal, view
onERC1155Received	Handles the receipt of a single ERC1155 token type.	external, returns(bytes4)
onERC1155BatchReceived	Handles the receipt of multiple ERC1155 token types.	external, returns(bytes4)

Table B.6: Converter Functions

Attribute	Description	Type
rewardDebt	Amount of UNICs owed but not yet distributed to a user.	uint256
lpToken	Address of the LP token contract for each pool.	IERC20
allocPoint	Allocation points for distributing UNICs in each pool.	uint256
uToken	Address of the associated uToken, if applicable.	address
unic	The UNIC token used for rewards.	Unic
devaddr	Address for developer incentives.	address
mintRateMultiplier	Multiplier for UNIC mint rate calculation.	uint256
totalAllocPoint	Total allocation points across all pools.	uint256
startBlock	Block number when UNIC mining begins.	uint256

Table B.7: UnicFarm Attributes

Function	Description	Modifiers
deposit	Stake LP tokens in a pool to earn UNIC rewards.	None
withdraw	Withdraw LP tokens from a pool.	None
updatePool	Update the state of a pool for rewards calculation.	None
setMintRules	Configure minting rules for UNIC tokens.	onlyOwner
add	Add a new LP token pool with specified parameters.	onlyOwner
set	Updates the allocation points for a given pool.	onlyOwner
emergencyWithdraw	Allows users to immediately withdraw their LP tokens without rewards.	None
setStartBlock	Sets the starting block for UNIC mining.	onlyOwner

Table B.8: UnicFarm Functions

Attribute	Description	Type
unic	The Unic token contract used for staking.	IERC20

Table B.9: UnicGallery Attributes

Function	Description	Modifiers
enter	Stake UNIC tokens in exchange for xUnic shares.	None
leave	Withdraw staked UNIC tokens by redeeming xUnic shares.	None

Table B.10: UnicGallery Functions

Attribute	Description	Type
factory	The UnicSwap factory used for trading operations.	IUnicSwapV2Factory
bar	The xUNIC token address, representing staked UNIC tokens.	address
unic	The UNIC token's contract address.	address
weth	The wrapped Ethereum (WETH) token's contract address.	address

Table B.11: UnicPumper Attributes

Function	Description	Modifiers
convert	trade a pair of tokens for UNIC, utilizing the UnicSwap factory.	None
_toWETH	Converts a specified token to WETH.	Internal
_toSUSHI	Converts WETH to UNIC, facilitating reward distribution.	Internal

Table B.12: UnicPumper Functions

Attribute	Description	Type
feeTo	The address where trading fees are directed.	address
feeToSetter	The address with the authority to set the 'feeTo' address.	address
getPair	A mapping of token pairs to their corresponding liquidity pool addresses.	mapping
allPairs	An array of all liquidity pool addresses created by the factory.	address[]

Table B.13: UnicSwapV2Factory Attributes

Function	Description	Modifiers
allPairsLength	Returns the total number of pairs created.	None
createPair	Creates a new liquidity pair and pool.	None
setFeeTo	set the address to receive trading fees.	None
setFeeToSetter	Allows modification of the ‘feeToSetter‘ address.	None

Table B.14: UnicSwapV2Factory Functions

Attribute	Description	Type
admin	The address with administrative privileges.	address
pendingAdmin	The address set to become the new admin.	address
delay	The mandatory delay before executing transactions.	uint
MINIMUM_DELAY	The minimum allowable delay.	constant uint
MAXIMUM_DELAY	The maximum allowable delay.	constant uint
GRACE_PERIOD	The period after delay expiry within which execution can occur.	constant uint

Table B.15: Timelock Attributes

Function	Description	Modifiers
setDelay	Sets the transaction execution delay.	None
acceptAdmin	transfer the admin role to the pendingAdmin.	None
setPendingAdmin	Sets a new pendingAdmin.	None
queueTransaction	Queues a transaction for future execution.	None
cancelTransaction	cancel a queued transaction.	None
executeTransaction	Execute a queued transaction post-delay.	None

Table B.16: Timelock Functions

Attribute	Description	Type
proposalCount	The total number of proposals made.	uint
timelock	Reference to the Timelock contract for execution delays.	TimelockInterface
unic	The UNIC governance token contract.	Unic
guardian	Address of the guardian with special administrative privileges.	address
quorumVotes	Number of votes required for quorum.	function
proposalThreshold	Minimum votes required to make a proposal.	function
proposalMaxOperations	Maximum operations allowed in a proposal.	function
votingDelay	Delay before voting on a proposal starts.	function
votingPeriod	Duration for which voting is open on a proposal.	function

Table B.17: GovernorAlpha Attributes

Appendix C

NFTX Smart Contracts

Function	Description	Modifiers
propose	Initiate a new governance proposal.	None
queue	Prepare a successful proposal for execution after a delay.	None
execute	Execute a queued proposal.	None
cancel	Cancel a proposal.	None
castVote	Cast a vote on a proposal.	None
castVoteBySig	Casts a vote on a proposal using a signature.	None
_acceptAdmin	Transfers Timelock admin role.	None
_abdicate	Guardian abdicates their role.	None
_queueSetTimelockPendingAdmin	Queues the setting of a new Timelock pending admin.	None
_executeSetTimelockPendingAdmin	Executes the setting of a new Timelock pending admin.	None

Table C.1: GovernorAlpha Functions

Attribute	Description	Type
vaultId	Unique identifier for the vault	uint256
manager	Address of the vault's manager	address
assetAddress	Address of the NFT asset	address
vaultFactory	Factory contract for creating new vaults	INFTXVaultFactory
eligibilityStorage	Contract for checking NFT eligibility	INFTXEligibility
is1155	Indicates if the vault is for ERC1155 NFTs	bool
enableMint	Flag to enable or disable minting feature	bool
enableRandomRedeem	Flag for enabling random redeem feature	bool
enableTargetRedeem	Flag for enabling targeted redeem feature	bool
enableRandomSwap	Flag for enabling random swaps of NFTs	bool
enableTargetSwap	Flag for enabling targeted swaps of NFTs	bool
randNonce	Nonce for random number generation	uint256
holdings	Set of token IDs held in the vault	EnumerableSetUpgradeable.UintSet
quantity1155	Mapping of token IDs to their quantities for ERC1155	mapping(uint256 => uint256)

Table C.2: NFTXVaultUpgradeable most Important Attributes

Function	Description	Modifiers
mint	Mints new tokens in exchange for NFTs	nonReentrant, onlyOwnerIfPaused
mintTo	Mints tokens to a specified address in exchange for NFTs	nonReentrant, onlyOwnerIfPaused
redeem	Redeems NFTs for tokens	nonReentrant, onlyOwnerIfPaused
redeemTo	Redeems NFTs to a specified address	nonReentrant, onlyOwnerIfPaused
swap	Swaps one NFT for another within the same vault	nonReentrant, onlyOwnerIfPaused
swapTo	Swaps NFTs to a specified address within the same vault	nonReentrant, onlyOwnerIfPaused
flashLoan	Executes a flash loan of tokens	nonReentrant, onlyOwnerIfPaused
setVaultMetadata	Sets metadata for the vault	onlyPrivileged
setVaultFeatures	Sets various features of the vault like minting, redeeming	onlyPrivileged
setFees	Sets various fees for vault operations	onlyPrivileged
disableVaultFees	Disables fees for the vault	onlyPrivileged
deployEligibilityStorage	Deploys a contract for eligibility checks	onlyPrivileged

Table C.3: NFTXVaultUpgradeable most Important Functions

Attribute	Description	Type
zapContract	Previously used, now for compatibility.	Address
feeDistributor	Address for fee distribution management.	Address
eligibilityManager	Manages eligibility for NFTs within vaults.	Address
vaults	Array storing all vault addresses.	Address[]
excludedFromFees	Addresses excluded from paying fees.	Mapping
_vaultFees	Stores fee settings for each vault.	Mapping
factoryFees	Stores default fee settings for the factory.	Multiple uint64

Table C.4: NFTXVaultFactoryUpgradeable Attributes

Function	Description	Modifiers
createVault	Creates a new vault with specified parameters.	onlyOwnerIfPaused
setFactoryFees	Sets default fees for various vault operations.	onlyOwner
setVaultFees	Sets fees for a specific vault.	None
disableVaultFees	disable fees for a specific vault.	None
setFeeDistributor	Sets the fee distributor address.	onlyOwner
setFeeExclusion	Excludes specific addresses from fees.	onlyOwner
setEligibilityManager	Sets the eligibility manager address.	onlyOwner
vaultFees	Retrieves fee settings for a specific vault.	None
isLocked	Checks if specific functionality is locked.	None
vaultsForAsset	Retrieves all vaults for a specific asset.	None
vault	Retrieves a vault address by its ID.	None
allVaults	Returns all vault addresses.	None
numVaults	Returns the total number of vaults.	None
deployVault	deploy a new vault contract.	None

Table C.5: NFTXVaultFactoryUpgradeable Functions

Attribute	Description	Type
modules	Array storing information about eligibility modules, including implementation, target asset, and name.	EligibilityModule[]

Table C.6: NFTXEligibilityManager Attributes

Function	Description	Modifiers
_NFTXEligibilityManager_init	Initializes the contract setting the initial owner.	initializer
addModule	Adds a new eligibility module to the modules array.	onlyOwner
updateModule	update an existing module's implementation address.	onlyOwner
deployEligibility	Deploys an eligibility module for a specific module index, using the configuration data.	None
allModules	Returns an array of all registered eligibility modules.	None
allModuleNames	Returns an array of names of all registered eligibility modules.	None

Table C.7: NFTXEligibilityManager Functions

Attribute	Description	Type
nftxVaultFactory	Reference to the NFTX vault factory for interaction.	INFTXVaultFactory
rewardDistTokenImpl	Manages reward distribution.	IRewardDistributionToken
stakingTokenProvider	Supplies staking tokens for vaults.	StakingTokenProvider
vaultStakingInfo	Maps vault IDs to staking pools.	Mapping

Table C.8: NFTXLPStaking Attributes

Function	Description	Modifiers
addPoolForVault	Creates a staking pool for a vault.	onlyAdmin
setNFTXVaultFactory	Sets the NFTX vault factory, designed to be immutable post-initialization.	onlyOwner
updatePoolForVaults	Updates staking pools in batch.	None
receiveRewards	Handles the reception and distribution of rewards.	onlyAdmin
deposit	Allows users to deposit tokens into a staking pool.	None
exit	Enables users to exit a staking pool, claiming rewards.	None
claimRewards	Users can claim their staking rewards.	None
emergencyExit	Provides an emergency function for token withdrawal.	None

Table C.9: NFTXLPStaking Functions

Attribute	Description	Type
nftxVaultFactory	Reference to the NFTX Vault Factory.	INFTXVaultFactory
lpStaking	Address of the LP staking contract.	address
treasury	Treasury address for fund management.	address
allocTotal	Total allocation points for fee distribution.	uint256
feeReceivers	Array of fee receivers and their allocation points.	FeeReceiver[]
inventoryStaking	Address of the inventory staking contract.	address

Table C.10: NFTXSimpleFeeDistributor Attributes

Function	Description	Modifiers
distribute	Distribute fees to various receivers based on allocation points.	nonReentrant
addReceiver	Add a new fee receiver with allocation points.	onlyOwner
initializeVaultReceivers	Initializes fee receivers for a specific vault.	None
removeReceiver	Removes a fee receiver from the list.	onlyOwner
setTreasuryAddress	Sets the treasury address.	onlyOwner
pauseFeeDistribution	Pauses or unpauses fee distribution.	onlyOwner
rescueTokens	Allows the rescue of mistakenly sent tokens.	onlyOwner

Table C.11: NFTXSimpleFeeDistributor Functions