

Vulnerabilities of smart contracts and mitigation schemes: A Comprehensive Survey

Wejdene Haouari

*Department of Electrical Engineering
& Computer Science,
York University, ON, Canada*

Abdelhakim Senhaji Hafid

*Department of Computer Science
and Operational Research,
University of Montreal, QC, Canada*

Marios Fokaefs

*Department of Electrical Engineering
& Computer Science,
York University, ON, Canada*

Abstract—Ethereum smart contracts are highly powerful; they are immutable and retain massive amounts of tokens. However, smart contracts keep attracting attackers to benefit from smart contract flaws and Ethereum’s unexpected behavior. Thus, methodologies and tools have been proposed to help implement secure smart contracts and to evaluate the security of smart contracts already deployed. Most related surveys focus on tools without discussing the logic behind them; in addition, they assess the tools based on papers rather than testing the tools and collecting community feedback. Other surveys lack guidelines on how to use tools specific to smart contract functionalities. This paper presents a literature review combined with an experimental report that aims to assist developers in developing secure smart contracts, with a novel emphasis on the challenges and vulnerabilities introduced by NFT fractionalization by addressing the unique risks of dividing NFT ownership into tradeable units called fractions. It provides a list of frequent vulnerabilities and corresponding mitigation solutions. In addition, it evaluates the community’s most widely used tools by executing and testing them on sample smart contracts. Finally, a comprehensive guide on how to implement secure smart contracts is presented.

Index Terms—Blockchain, Ethereum, Smart contracts, Formal verification, Semantic verification, Fuzzing, Software security, and Software quality.

1. Introduction

Blockchain technology has exploded in popularity over the years due to its immutability, security, and transparency in a permissionless and decentralized environment. One of the well-known and used implementations is Ethereum. As of 31 May 2023, the Ethereum cryptocurrency’s market capital surpassed \$220 billion, with millions of transactions being executed every day [1]. Ethereum uses Turing Complete blockchain technology [2], allowing developers to implement smart contracts. Smart contracts are executable programs stored in the blockchain written primarily in solidity [3]. Using smart contracts facilitates the execution of pre-defined terms without consulting third parties in an anonymous, transparent, and tempered manner.

The Ethereum blockchain is the foundation for numerous financial projects. Examples include (a) platforms for

decentralized finance (DeFi), which enable users to access various financial services like lending and borrowing, trading, and insurance without intermediaries. (b) Stablecoins, cryptocurrencies whose value is tied to that of a fiat currency or other asset, like gold or the US dollar; (c) Security token offerings (STOs), which are virtual assets that simulate ownership of a physical asset like a stock or piece of real estate. (d) Non-fungible tokens (NFTs), which are unique digital assets that might signify ownership of a physical or digital object, like a work of art or a collectible, (e) Amidst this variety, NFT fractionalization emerges as a noteworthy trend, dividing ownership of NFTs into smaller, more accessible units, thus broadening participation in the digital asset market. This value attracts attackers to exploit different vulnerabilities related to implementing smart contracts to steal cryptocurrencies or tamper with assets. For instance, a recent attack was on May 2023 where Level Finance Exchange announced the loss of more than 214,000 \$LVL tokens, approximated to be \$1.01 million [4], caused by an attack on its smart contract by manipulating a recursive calling vulnerability. There are various vulnerabilities and causes, such as arithmetic overflows, reentrancy, inadequate randomness, calling an unknown third-party contract code, and failing to check the return status of external calls. The runtime environment that supports contract code execution provides more attack points. One example is when malevolent miners pick and choose which transactions are included in a mined block or in what order they are included.

Since vulnerabilities in smart contracts can result in significant financial loss, multiple tools are proposed to identify vulnerabilities in Solidity smart contracts. Detection methods include (a) Symbolic execution when the program is abstractly executed to cover various possible inputs; (b) Fuzzy testing by injecting several invalid inputs; (c) Taint analysis by checking an input flow; and (d) Formal verification, when the behavior of the smart contract is checked mathematically using a formal model.

In this paper, we conduct a literature review and an experimental report. We present the most common vulnerabilities in Solidity smart contracts, including those posed by the fractionalization of NFTs and the corresponding mitigation schemes. We also overview and compare the most popular schemes and tools used to detect smart contract vulnerabilities. Finally, we propose a set of guidelines for

auditing smart contracts.

In this survey, we searched 2 popular databases, Engineering Village and Scopus, and web articles written by the Ethereum community Concerning vulnerabilities in Solidity smart contracts, mitigation tools, and guidelines. Moreover, we experimentally studied 5 tools widely used to detect vulnerabilities in smart contracts, namely Oyente [5], Slither [6], Mythril [7], Manticore [8] and Echidna [9].

The rest of this paper is organized as follows: Section 2 outlines the methodology we used to produce this survey. Section 3 presents the most common vulnerabilities of smart contracts and the corresponding mitigation schemes. Section 4 presents standard methodologies for detecting smart contract vulnerabilities. Section 5 introduces the most valuable tools to detect vulnerabilities in smart contracts. Section 6 presents a set of guidelines for auditing smart contracts. Section 7 presents an overview of existing related surveys. Finally, Section 8 concludes the paper.

2. 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) [10] for data preparation; This technique incorporates both gray literature and white literature. Gray literature includes blogs, videos, and forums; it is usually written by practitioners from industry and academia. 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 helps provide a complete overview. Moreover, the feedback provided by smart contract developers is crucial in selecting the most useful tools.

As for the search strategy, we followed a protocol presented by Kitchenham for systemic reviews [11]. This protocol consists of three phases: (a) Define a search string;

```
(( 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*) )
```

Figure 1: Used search string

(b) Use the string in search engines; and (c) Select literature based on predefined inclusion and exclusion criteria.

We defined a search string presented in Listing 1 to collect the following resources: (a) Review articles about smart contract security; (b) Papers that discuss vulnerabilities in Ethereum smart contracts; (c) Papers that cover detection methodologies or mitigation schemes of smart contract vulnerabilities; and (d) Papers about tools that detect vulnerabilities in smart contracts. We have only included papers containing the keywords in the subject, title, or abstract. The term "opcode" refers to the basic instructions carried out 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. This means it can resolve any computational problem given enough time and resources. Both those terms are used to represent smart contracts.

Figure 2 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 1. As for gray literature, we focused on blog posts found within the first eight pages of Google search results. The search strings include "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 reputation of the publisher.
- The author's expertise in smart contract security, such as job position.
- The content is clearly stated with supported arguments.

3. Vulnerabilities

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

3.1. Reentrancy

3.1.1. 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

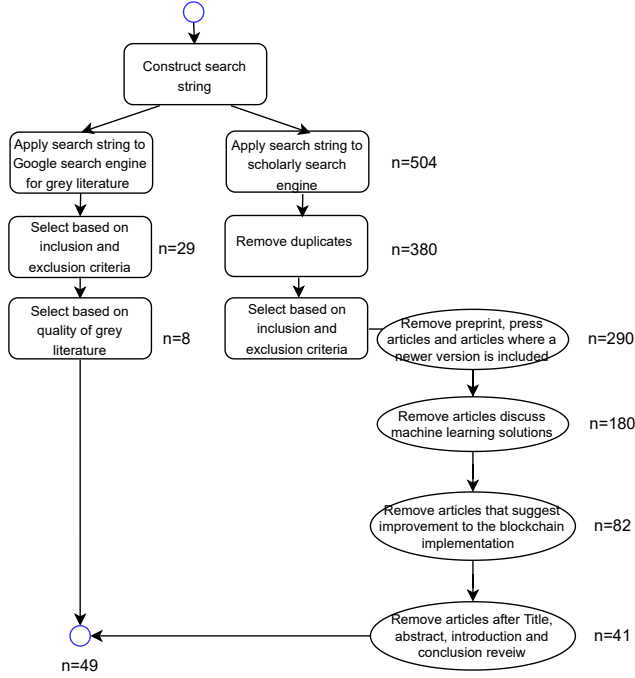


Figure 2: Articles identification, exclusion and inclusion methodology.

Criteria	Description
Inclusion Criteria	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - 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 training models, accuracy, etc. - Studies that are based on the improvement of blockchain infrastructure rather than smart contract programming - Study that is an older version of another paper that is previously been considered.

TABLE 1: Exclusion/inclusion criteria

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).

3.1.2. Implications on NFT Fractionalization. Reentrancy vulnerabilities have significantly impacted decentralized finance (DeFi) protocols, illustrating potential risks for fractionalized NFT platforms. For instance, the dForce DeFi Protocol Hack [12] in February 2023, where an attacker exploited a reentrancy vulnerability in the 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 [13] in August 2021 and the Siren Protocol Hack [14] 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.

3.1.3. Protection Measures. It is recommended to use *send()* and *transfer()* methods instead of the general *call()* method to transfer money. They are deemed safer because they have a gas limit of 2,300 gas. At the same time, the method *call()* does not have a gas limit and forwards the remaining gas to the target address. The gas limit prohibits the target contract from making costly external function calls. The *checks-effects-interactions* pattern [15] is the most reliable approach to prevent reentrancy attacks. This pattern defines how the code of a function should be organized to minimize undesirable side effects and execution behaviors. The programmer needs to include all checks, which usually consist of *assert()* and *require()* modifiers. If these checks pass, the function will resolve all of the effects of the contract (e.g., balance update). The function should only communicate with other contracts once all state changes have been updated. Even if an attacker performs a recursive call to the initial function, the user cannot exploit the state of the contract since external functions are called last.

Another protection mechanism against reentrancy attacks is the use of mutex. A mutex locks down the contract state. Mutex allows only one execution of a critical code section at a time using a lock mechanism. Only the lock’s owner can modify it. When an attacker tries to perform a reentrancy attack, the previous call will lock the state, and the balance can not be updated. Mutex must be treated with care to ensure the lock can be released. *ReentrancyGuard* is OpenZeppelin’s own mutex implementation [16]. This library offers a *nonReentrant* modifier that secures an external function with a mutex.

3.1.4. Example. Listing 3 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

```

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;
    }
}

```

Figure 3: Simplified contract with reentrancy vulnerability

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.

To address this vulnerability, the *ReentrancyGuard* utility from the OpenZeppelin security library can be integrated to prevent recursive calls, as demonstrated in Listing 4. 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.

3.2. Front Running

3.2.1. Description. Front-running, also known as transaction ordering dependency, occurs when the execution logic depends on the order of the submitted transactions. The miner [17] determines the order of transactions in Ethereum. The transaction is visible to the network before being executed. The participants can exploit this visibility by sending transactions with a higher gas price to be included first.

3.2.2. 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 significant transaction increases their value or, conversely,

```

pragma solidity ^0.8.0
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;
    }
}

```

Figure 4: Simplified contract that mitigates reentrancy vulnerability

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 shareowners, front-runners could strategically insert transactions to claim a disproportionate share of the distributions, undermining the fairness of the process.

3.2.3. Protection Measures. The best solution to protect against front-running vulnerability is to remove the advantage of transaction ordering from the application. A 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.

3.2.4. Example. Listing 5 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.

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. This is followed by a reveal phase where the transaction details are disclosed as shown in Listing 6.


```

contract FractionalNFTToken {
    uint public tokenPrice;
    uint public availableTokens;
    address owner;

    function buyTokens(uint quantity) public payable
    ↪ {
        require(msg.value >= quantity * tokenPrice,
        ↪ "Insufficient payment");
        require(quantity <= availableShares, "Not
        ↪ enough shares available");
        availableShares -= quantity;
        // Transfer logic for shares not shown for brevity
    }
}

```

Figure 5: Simplified contract with Front Running vulnerability

```

contract SecureFractionalNFTTokens is ReentrancyGuard
↪ {
    uint public tokenPrice;
    uint public availableTokens;
    address owner;
    mapping(bytes32 => bool) public commitments;
    uint public revealEndTime;
    bool public saleStarted;
    function commitToBuy(bytes32 commitment) public {
        require(block.timestamp < revealEndTime,
        ↪ "Commit phase is over");
        commitments[commitment] = true;
    }
    function revealBuy(uint quantity, uint maxPrice,
    ↪ bytes32 nonce) public payable nonReentrant {
        require(saleStarted & block.timestamp >=
        ↪ revealEndTime, "Reveal phase not
        ↪ started");
        bytes32 commitment =
        ↪ keccak256(abi.encodePacked(msg.sender,
        ↪ quantity, maxPrice, nonce));
        require(commitments[commitment], "Invalid
        ↪ commitment");
        require(maxPrice >= tokenPrice, "Share price
        ↪ exceeded max price");
        require(msg.value >= quantity * tokenPrice,
        ↪ "Insufficient payment");
        require(quantity <= availableTokens, "Not
        ↪ enough shares available");
        availableTokens -= quantity;
        // Logic to handle purchase and transfer token
    }
}

```

Figure 6: Simplified contract that mitigates Front Running vulnerability

3.3. Arithmetic

3.3.1. 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) [18]. These vulnerabilities can cause smart contracts to behave unpredictably, potentially leading to significant security breaches or financial losses.

3.3.2. Implications on NFT Fractionalization. In the context of NFT fractionalization, arithmetic vulnerabilities pose

```

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;
    }
}

```

Figure 7: Simplified contract with Arithmetic vulnerability

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 [19], 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 [20].

3.3.3. 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 [18]. 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.

3.3.4. Example. Listing 7 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.

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 to further safeguard the integrity of fractional token calculations and transfers. An improved version of the smart contract is presented in Listing 8.

```

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;
    }
}

```

Figure 8: Simplified contract that mitigates arithmetic vulnerability

3.4. Mishandled Exceptions

3.4.1. 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 [21]. 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 [22].

3.4.2. 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.

```

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}("");
        }
    }
}

```

Figure 9: 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");
        }
    }
}

```

Figure 10: Simplified contract that mitigates mishandled exceptions

3.4.3. Example. The example 9 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 10, we use a safer approach to distribute proceeds by checking the success of each payment and reverting the transaction if a payment fails.

3.5. Code Injection via delegatecall

3.5.1. 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 dynamically enables a

smart contract to load code from another smart contract at runtime. 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.

3.5.2. 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 [23], 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.

3.5.3. 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.

3.5.4. Example. Listing 11 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 the data. If *logicContract* points to a malicious contract, it could lead to unintended alterations in its state, including token ownership manipulation.

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 12 illustrates these changes.

3.6. Randomness Using Block Information

3.6.1. Description. In blockchain-based applications, certain functionalities might require randomness—for instance, distributing a rare NFT fractionally among participants or

```
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");
    }
}
```

Figure 11: Simplified contract with Code Injection via *delegatecall* vulnerability

```
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");
    }
}
```

Figure 12: Simplified contract that mitigates Injection via *delegatecall* vulnerability

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 [24]. 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,

```

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;
    }
}

```

Figure 13: Simplified contract with Randomness Using Block Information vulnerability

therefore, collect the reward.

3.6.2. 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.

3.6.3. 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 [25] or other external sources that provide verified randomness robust solution.

Alternatively, cryptographic commitment schemes [26], exemplified by solutions like RANDOA [27], 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.

3.6.4. Example. Listing 13 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.

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 14 inherits from VRFConsumerBase, provided by Chainlink [28], to securely request and receive verified random numbers. It uses the

```

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;
        }
    }

```

Figure 14: Simplified contract with Randomness Using Block Information vulnerability

Chainlink VRF (Verifiable Random Function) to ensure that participants, including miners, cannot influence randomness in determining the distribution of NFT fractions.

3.7. Other Vulnerabilities & Summary

Table 2 presents the papers that include each vulnerability. We notice that the most mentioned vulnerability is reentrancy, whereas code injection is the least discussed.

	Reentrancy	Front Run- ning	Arithmetic	Mishandled exceptions	Code Injec- tion	Randomness
[29]	X					
[30]	X		X			X
[31]	X			X	X	X
[32]	X	X				X
[33]	X					
[34]	X		X	X		X
[24]	X		X	X	X	X
[35]	X	X		X	X	X
[36]	X			X		X
[37]	X	X		X		X
[5]	X	X		X		X
[38]	X					
[6]	X					
[18]	X	X	X			
[39]	X	X		X		
[40]		X				X
[41]			X			
[42]			X			
[43]					X	

TABLE 2: Coverage of vulnerabilities by each reference

The Smart Contract Weakness Classification Registry (SWC) [44] 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. [45] 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.

4. 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.

4.1. 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 [46]. 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 [47]. This section presents some static analysis techniques, namely Control Flow Graph, Taint Analysis, and Symbolic Analysis.

4.1.1. Control Flow Graph.

The Control Flow Graph (CFG) is a directed graph describing the control flow. The flow of execution of source code. A graph node denotes a basic block with no jumps; directed edges represent jumps from one block to another. If a node only has an exit edge, it is referred to as an *entry*. And if it has only an entry edge, it is referred to as an *exit* block. In Figure 15, Block 1448 is an entry block, and blocks 1451 and 1452 are exit blocks.

Figure 16 shows the standard process to create CFG in the context of Ethereum. The first step is **Parsing bytecode**. Typically, the algorithm will extract the compiler version from the metadata; then, it will parse the remaining bytecode (without the metadata) into opcodes [48]. The parsing stage is straightforward because each bytecode's two characters represent one opcode (0x04 represents DIV). The complete list of opcodes can be found in the Ethereum yellow paper [49]. The second step is **Identification of basic blocks**. A basic block is a set of opcodes that run in succession between a jump target and a jump instruction, with no other instructions interrupting the control flow. To determine the basic blocks, we need to identify the opcodes that alter the execution of the control flow; some examples are presented in Table 3. The JUMPDEST instruction starts a new basic block, whereas the other opcodes terminate blocks. Each block is identified by an offset representing its first opcode's

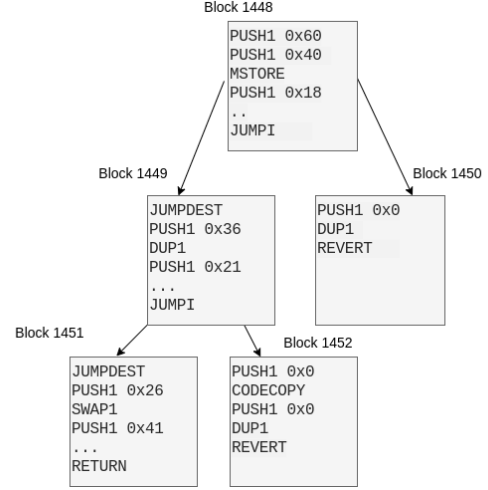


Figure 15: Example Control Flow Graph

Operation	Opcode	Description
Flow	JUMP	Alters the program counter
Operations	JUMPI	Alters the program counter
	JUMPDEST	Conditionally alters the program counter
		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 3: Opcodes that alter execution [49]

position in the bytecode. The last step will be computing the edge by checking the destination offset; this step is not trivial as the destination is not an opcode parameter. In the literature, there are two types of jumps: (a) A *pushed jump* is a JUMP followed by a PUSH opcode, allowing the target offset to be determined simply by glancing at the data in the previous PUSH opcode; and (b) *Orphan jump* is not followed by a PUSH; indeed, its destination is not computed directly [50]. The computation of orphan jumps depends on each implementation and is critical for the completeness of the graph.

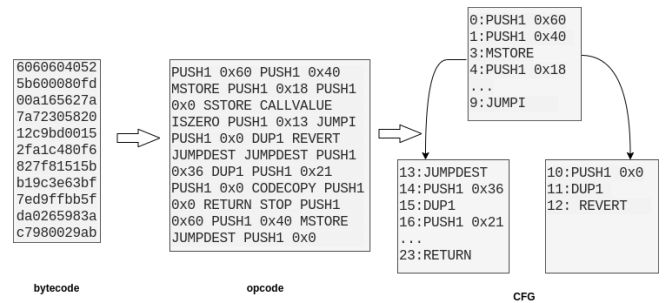


Figure 16: Process of generating CFG [49]

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

TABLE 4: Opcodes that allow data insertion [49]

4.1.2. Taint Analysis.

Taint Analysis aims to detect input variables where data comes from an untrusted source that could be controlled by an attacker (e.g., environmental data or function parameters). These data inputs are *tainted* by the taint analysis tool and then traced back to potentially sensitive functions, such as security checks or storage access, often known as *sinks*. If the tainted variable flows into a sink, it is marked as a vulnerability [41].

In the context of Ethereum smart contracts, this technique is usually used after generating CFG. It exploits the opcode that an attacker may employ to insert data into the control flow; it can be divided into block, environmental, and flow information. Table 4 shows an example of an opcode that allows data insertion can be found in.

An example of taint analysis is to detect arithmetic bugs. The first step is to define the possible entrusted inputs (source), such as CALLDATALOAD and SLOAD. The affected memory, storage, or stack location is tagged. The next step is to check whether the flow of tagged input contains arithmetic operations such as ADD, MUL, and SUB with no catch mechanism (sink). For example, a catch mechanism is reverted in the case of DIV with 0. If the tagged input falls into a sink, it will be marked as a possible vulnerability.

4.1.3. Symbolic Analysis.

Symbolic execution is a way of abstracting the execution of a program such that it can span numerous pathways through the code. The program is run with *symbols* as inputs, and the outputs are expressed in the form of the symbolic inputs. Each symbolic path has a condition; it is a formula created by collecting constraints that must be satisfied by those inputs for the execution to continue on that path. If the condition is unsatisfiable, the path is infeasible; otherwise, the path is feasible [51]. Symbolic analysis is usually paired with CFGs. To find exploits, most tools employ the Satisfiability Modulo Theories (SMT) [52] solver to check whether the symbolic output is feasible or provide a counterexample if it is not.

4.2. Dynamic Analysis

”dynamic analysis” refers to observing code while 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.

4.2.1. 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. [53] 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.

4.3. 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 first need to provide specifications. A program specification is an unambiguous definition of the program’s purpose 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 [54]. 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, theorem-proving algorithms are usually used, such as Coq [55] and Isabelle/HOL [54].

There are a few contributions that propose languages to write formal specifications. Permenev et al. [56] 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 [57], which is a specification for Ethereum Virtual Machine (EVM) programs.

4.4. 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 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 [58].

Vulnerabilities	References
Static analysis	[32] [5] [41] [39] [59] [6] [60] [18] [42] [56] [61]
Dynamic analysis	[62] [9] [31] [63] [64] [65] [8]
Formal verification	[66] [60] [54] [32] [61]

TABLE 5: Detection methods References

5. 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 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.

5.1. Oyente

5.1.1. Description. Luu et al. propose Oyente [5], the first symbolic execution tool for Ethereum smart contracts. Oyente was the basis of several tools developed later [62] [41]. It has four components: (a) *CFGBuilder*: It constructs CFG of the smart contract; (b) *Explorer*: It takes as input the Ethereum state. It has a loop that runs a state and then executes an instruction on the output of that state. The loop continues until no state remains or a timeout is reached. The output is a symbolic trace. *Explorer* determines the infeasible trace by querying the Z3 SMT solver [84]; (c) *CoreAnalysis*: It targets predefined vulnerabilities by looking for patterns on the symbolic trace; (d) *Validator*: It queries Z3 solver with traces flagged as vulnerable to reduce false positives cases.

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

#	Tool	Detection Technique	Last update
1	Slither [6] [67]	Static Analysis	May 2023
2	MythX [68]	static, dynamic & Symbolic Execution	Not Public
3	Mythril [7] [69]	Symbolic Execution	May 2023
4	Echidna [9] [70]	Fuzzer	Apr 2023
5	Manticore [8] [71]	Symbolic Execution	June 2022
6	Securify [39] [72]	Static Analysis	Sep 2021
7	KEVM [60] [73]	Static Analysis	Apr 2022
8	Smartcheck [18] [74]	Static Analysis	Dec 2019 deprecated
9	MadMax [42] [75]	static analysis	Jun 2021
10	Vertigo [76] [77]	Mutation Testing	Feb 2021
11	EtherSolve [38] [78]	CFG Extraction	Nov 2021
12	Octopus [79]	Symbolic Execution	Nov 2020
13	Oyente [5] [80]	Symbolic Execution	Nov 2020 deprecated
14	ERC20 Verifier [81]	Verify ERC20 Compatibility	Nov 2019
15	Solgraph [82]	CFG Extraction	Jan 2019
16	Osiris [41] [83]	Symbolic Execution	Sep 2018

TABLE 6: Ethereum Smart Contract Vulnerability Detection Tools

5.1.2. 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 [85]. 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 17, highlight an Integer Overflow vulnerability and the execution trace where this issue is detected.

5.1.3. 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;

```

WARNING:root:You are using evm version 1.0.2, 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 = 115792089237316195423570985008687907853269984665640564039457584007913129639935
INFO:symExec: ===== Analysis Completed =====

```

Figure 17: Oyente Output

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 [86] and Osiris [41]. These tools have built upon Oyente’s foundational techniques, such as constructing Control Flow Graphs (CFGs), to offer enhanced accuracy in detecting vulnerabilities and orphan paths within smart contracts.

5.2. Slither

5.2.1. Description. In their work, Feist et al. [6] unveiled a static analysis instrument known as Slither. This tool is aimed at spotting weaknesses, enhancing code efficiency, and augmenting the understanding of code. The procedure it employs is in multiple stages: (a) *Data Retrieval*: Initially, it creates the Abstract Syntax Tree of the contract’s source code using the Solidity compiler, yielding some information like the Control Flow Graph and contract inheritance; (b) *Conversion into SlitherIR*: The source code of the smart contract is then converted into an internal representation language referred to as SlithIR; and (c) *Examination of the Code*: This phase involves determining the variables being read and written as well as their types. Furthermore, it detects unsecured functions where potentially harmful addresses can execute high-level operations. During this *Examination of the Code* stage, it also calculates data dependency and marks variables that rely on user input as tainted.

Slither is proficient in uncovering 70 bug varieties, encompassing self-destructive smart contracts, code injection via delegatecall, frozen ether, and Reentrancy vulnerabilities [87].

5.2.2. Execution Example. Using Slither to identify potential security vulnerabilities, we analyzed the *FractionalNFTMarket* smart contract in Listing 18. The analysis results are presented in Figure 19. Slither has successfully detected a reentrancy vulnerability within the *sellShares* function, which could allow an attacker to exploit the contract by

```

contract FractionalNFTMarket {
    address public nftOwner;
    uint256 public totalShares = 100;
    mapping(address => uint256) public sharesOwned;
    uint256 public pricePerShare = 0.01 ether;

    constructor() {
        nftOwner = msg.sender;
        sharesOwned[msg.sender] = totalShares;
    }

    function sellShares(uint256 shares) public {
        require(sharesOwned[msg.sender] >= shares,
            "Not enough shares owned");

        (bool sent, ) = msg.sender.call{value: shares
            * pricePerShare}("");
        require(sent, "Failed to send Ether");

        sharesOwned[msg.sender] -= shares;
        sharesOwned[nftOwner] += shares;
        emit ShareSold(msg.sender, shares);
    }
}

```

Figure 18: FractionalNFTMarket Smart Contract

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.

```

Reentrancy in FractionalNFTMarket.sellShares(uint256) (FractionalNFTMarket.sol#29-38):
  External calls:
  - (sent) = msg.sender.call{value: shares * pricePerShare}() (FractionalNFTMarket.sol#32)
  State variables written after the call(s):
  - sharesOwned[msg.sender] = shares (FractionalNFTMarket.sol#35)
  FractionalNFTMarket.sharesOwned (FractionalNFTMarket.sol#7) can be used in cross function reentrancies:
  - FractionalNFTMarket.buyShares(uint256) (FractionalNFTMarket.sol#19-26)
  - FractionalNFTMarket.constructor() (FractionalNFTMarket.sol#13-16)
  - FractionalNFTMarket.sellShares(uint256) (FractionalNFTMarket.sol#29-38)
  - FractionalNFTMarket.sharesOwned (FractionalNFTMarket.sol#7)
  - sharesOwned[nftOwner] += shares (FractionalNFTMarket.sol#36)
  FractionalNFTMarket.sharesOwned (FractionalNFTMarket.sol#7) can be used in cross function reentrancies:
  - FractionalNFTMarket.buyShares(uint256) (FractionalNFTMarket.sol#19-26)
  - FractionalNFTMarket.constructor() (FractionalNFTMarket.sol#13-16)
  - FractionalNFTMarket.sellShares(uint256) (FractionalNFTMarket.sol#29-38)
  - FractionalNFTMarket.sharesOwned (FractionalNFTMarket.sol#7)
Reference: https://github.com/crytic/slither/wiki/detector-documentation#reentrancy-vulnerabilities
INFO:Detectors:
Reentrancy in FractionalNFTMarket.sellShares(uint256) (FractionalNFTMarket.sol#29-38):
  External calls:
  - (sent) = msg.sender.call{value: shares * pricePerShare}() (FractionalNFTMarket.sol#32)
  Event emitted after the call(s):
  - ShareSold(msg.sender, shares) (FractionalNFTMarket.sol#37)
Reference: https://github.com/crytic/slither/wiki/detector-documentation#reentrancy-vulnerabilities-3

```

Figure 19: Slither Analysis Output for FractionalNFTMarket Contract

Slither also provides a list of helpful information, including a human-readable summary of the scanned contract, inheritance graph, and CFG of each function; the complete list can be found in the GitHub repository [67].

5.2.3. 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 [88].

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 [89], a premium service, can be utilized for a more comprehensive analysis. Crytic extends Slither’s capabilities by identifying an additional 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.

5.3. Mythril

5.3.1. Description. Slithe Durieux et al. [7] developed a tool known as Mythril to identify potential weaknesses in smart contracts by utilizing symbolic execution. Mythril, a Python-based command-line tool, allows for interactive inspection of smart contracts. It produces a Control Flow Graph (CFG) and performs symbolic execution of EVM bytecode to restrict the search area. The tool can generate concrete values to exploit any detected vulnerabilities. To evaluate the feasibility of different paths, Mythril uses the Z3 SMT solver [84].

Mythril can discover fourteen distinct types of vulnerabilities, including Delegate Call To Untrusted Contracts, Dependence on Predictable Variables, Deprecated Opcodes, Ether Thief, Exceptions, External Calls, Integer Over/Underflow, Multiple Sends, Self-Destruction, State Change External Calls, Unchecked Return Values, User Supplied Assertions, and Arbitrary Storage Write and Arbitrary Jump [90].

5.3.2. Execution Example. The *FractionalNFTMarket.sol* smart contract, presented in Listing 18, has been analyzed using Mythril to uncover potential security issues. The resulting output is depicted in Figure 20. 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 the identification of vulnerabilities and 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.

```
==== State access after external call ====
SMT ID: 187
Severity: Medium
Contract: FractionalNFTMarket
Function name: sellShares(uint256)
PC address: 0x0
Estimated gas usage: 17536 - 93032
Head of persistent state following external call
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the caller is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callers from re-entering the contract in an intermediate state.
-----
In file: FractionalNFTMarket.sol:36
sharesOwned[owner] += shares
-----
Initial State:
Account: [CREATOR], balance: 0x0, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}
Transaction Sequence:
caller: [CREATOR], calldata: , decoded_data: , value: 0x0
caller: [ATTACKER], function: sellShares(uint256), tdata: 0x4b2a6e1200000000000000000000000000000000000000000000000000000000, decoded_data: (0,), value: 0x0
```

Figure 20: Mythril Output

5.3.3. Pros and Cons.

Mythril stands out for its accessibility, offering straightforward installation options via Docker or Python package manager, streamlining 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 [68], offers enhanced analysis power and can be seamlessly integrated into continuous integration/continuous deployment (CI/CD) pipelines [91].

5.4. Manticore

5.4.1. Description. Mossberg et al. [8] implemented a tool named Manticore to detect vulnerabilities in smart contracts. The primary aim of Manticore is to track inputs that kill a program, record instruction-level implementation, and provide Python API access to its analysis engine. Manticore uses symbolic execution to locate distinct computation paths in EVM bytecode. It discovers inputs that will stimulate these computation paths with the aid of the SMT solver Z3 [84]. It keeps track of the execution traces for each execution. Manticore converts Solidity code to bytecode for evaluation; then, it examines the traces for vulnerabilities, such as reentrancy and reachable self-destruct operations, reporting them in the source code context. This tool is developed by *TrailOfBits* [92].

Mythril detects ten types of vulnerabilities, namely, delegatecall, overflow, reentrancy, Use of potentially unsafe/manipulable instructions, Reachable external, Reachable self-destruct instructions, Uninitialized memory usage, Uninitialized storage usage, Enable INVALID instruction detection, and Unused internal transaction return values [93].

5.4.2. Execution Example. For our examination of Manticore, we used the *TokenSaleChallenge.sol* [85], 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; it takes less than a minute for the other tools using 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 contract’s reentrancy vulnerability, as shown in Figure 21.

5.4.3. 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

```
(m) wejdenn@jdenn:~/polytechnique/blockchainProject/mcore_42kbn$ cat global.summary
Global runtime coverage:
6f5c3c247fcdicefec3774a16119afb37565fa: 98.83%

Compiler warnings for TokenSaleChallenge:
tokensalechallenge.sol:7:33: warning: unused function parameter. Remove or comment out the variable name to silence this warning.
function TokenSaleChallenge(address _player) public payable {
    ^^^^^^^^^^^^^^^^^
```

Figure 21: Manticore Output

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.

5.5. Echidna

5.5.1. Description. Echidna is a specialized property-based fuzzing tool for smart contracts implemented by Grieco et al. [9] that takes inspiration from QuickCheck [94]. Echidna seeks to violate user-defined invariants that represent potential contract errors rather than just finding crashes [95]. The tester writes these invariants, and it is called *echidna property*: A particular Solidity function with no arguments that returns “true” on success and has a name that begins with “echidna.” Echidna reports any transactions that result in these properties returning false or error, essentially disclosing contract bugs [96]. Echidna employs various strategies to produce test inputs, including iterative feedback, structural constraints, and known input variation.

5.5.2. Execution Example. To test Echidna, we have adapted the *tokensalechallenge.sol* smart contract [85]. We created a derived contract *TestTokenSaleChallenge* that extends the original *TestTokenSaleChallenge* contract shown in Listing 22. 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.

Echidna generates its results directly to the console, as figure 23 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

```
pragma solidity ^0.5.0;

import "./TokenSaleChallenge.sol";

contract TestTokenSaleChallenge is TokenSaleChallenge
{

    function echidna_test_overflow() public view
    {
        returns (bool) {
            uint256 MAX_UINT = uint256(0);
            uint256 balance = balanceOf[msg.sender];
            return balance < MAX_UINT;
        }
    }

    function echidna_test_underflow() public view
    {
        returns (bool) {
            uint256 balance = balanceOf[msg.sender];
            return balance > 0;
        }
    }
}
```

Figure 22: TestTokenSaleChallenge Smart Contract

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.

```
Time elapsed: 4s
Workers: 0/1
Seed: 7576380395591485271
Calls/s: 12519
Total calls: 50077/50000

[Echidna 2.2.0]
Unique instructions: 441
Unique codehashes: 1
Corpus size: 8 seqs
New coverage: 15 ago

Chain ID: -
Fetched contracts: 0/0
Fetched slots: 0/0

Tests (2)
echidna_test_underflow: FAILED!
*no transactions made*
echidna_test_overflow: passing

Log (10)
[2023-06-14 16:04:47.76] [Worker 0] Test limit reached. Stopping.
[2023-06-14 16:04:45.85] [Worker 0] New coverage: 441 instr, 1 contracts, 8 seqs in corpus
[2023-06-14 16:04:44.11] [Worker 0] New coverage: 405 instr, 1 contracts, 7 seqs in corpus
[2023-06-14 16:04:44.05] [Worker 0] New coverage: 405 instr, 1 contracts, 6 seqs in corpus
[2023-06-14 16:04:43.97] [Worker 0] New coverage: 398 instr, 1 contracts, 5 seqs in corpus
[2023-06-14 16:04:43.72] [Worker 0] New coverage: 352 instr, 1 contracts, 4 seqs in corpus
[2023-06-14 16:04:43.71] [Worker 0] New coverage: 275 instr, 1 contracts, 3 seqs in corpus
[2023-06-14 16:04:43.65] [Worker 0] New coverage: 272 instr, 1 contracts, 2 seqs in corpus
[2023-06-14 16:04:43.64] [Worker 0] New coverage: 172 instr, 1 contracts, 1 seqs in corpus
[2023-06-14 16:04:43.63] [Worker 0] Test echidna_test_underflow falsified!

Campaign complete, C-c or esc to exit
```

Figure 23: Echidna Output

5.5.3. 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 cus-

tomized 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.

5.6. Summary

Table 7 presents a comparative analysis of Oyente, Slither, Mythril, Manticore, and Echidna. The comparison is based on the number of detected vulnerabilities, the methodology used, an academic or company tool, the code level, the required solidity version, and the availability of documentation.

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

TABLE 7: Qualitative comparison of selected tools

In conclusion, there are various tools that can be used to test smart contracts and identify a wide range of vulnerabilities. Static analysis tools are still the most popular because they are simple to use compared to (a) fuzzing, which necessitates a lot of resources to set up the test environment, and (b) formal verification, which necessitates expertise in contract specification writing.

6. Guidelines for secure smart contracts

Writing a secure and bug-free smart contract is crucial, as a small bug can lead to the loss of millions of dollars. In this section, we provide guidelines on how to write a secure smart contract [97]. These steps are summarized in Figure 24

The first step in testing a smart contract is to visualize its control flow correctly to have a global view and better understand the interactions between its different components. *Slither* is an exciting tool to use during this step since it has a variety of printers that can be used to visualize call graphs, contract inheritance relationships, modifiers called by each function, etc. *Solidity Visual Developer* is another interesting tool that could be integrated into Visual Studio Code [98]. This extension helps with semantic highlighting. It also generates a detailed class outline. This step is essential to spot critical functions that must be tested in the next stage.

Afterward, the smart contract should be subjected to automatic analysis. It is best to start with common bug

detection using *Mythril* and *Slither*; we recommend combining tools for better results. To target a certain critical function, it is best to use the *Manticore* tool as it allows the implementation of specific use cases. It is also crucial to employ dynamic analysis tools to decrease the number of false positives generated by static tools and to cover additional states. *Echidna* is an interesting ethereum fuzzing tool to use [9]. It is recommended to look for unique features in smart contracts. For instance, if the smart contract is an ERC token, it will be essential to use *Slither’s ERC Conformance* functionality to make sure that the smart contract conforms to the ERC standards put in place.

Because automated tools are unaware of the context for which the contract was produced, running more specific testing will increase the detection accuracy. To achieve this, *Echidna* and *Manticore* [99] allow for the definition of security properties in solidity. Indeed, these tools help check arithmetic operations, external interactions, and standardization. Whereas *Slither* allows the definition of properties with Slither Python APIs, it will enable checking for inheritance, variable dependencies, and access restrictions.

The third step will be formal verification, which ensures that the contract requirements are legitimate. This step, however, is challenging as it requires unique expertise (e.g., writing the formal specification using a specification language). Finally, the output of the preceding steps must be manually verified, and corrective actions based on patterns and mitigation techniques must be applied. After fixing the bugs, one has to redo the previous steps to increase the probability of a bug-free contract.

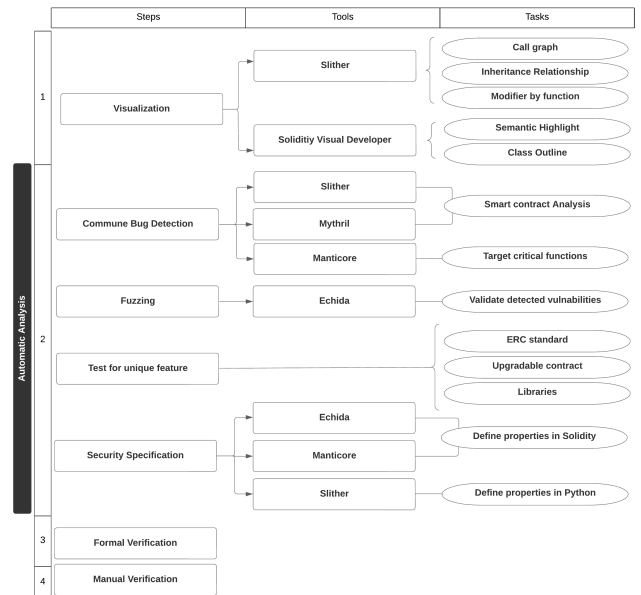


Figure 24: Smart contract auditing

7. Related Work

Several surveys covering smart contract vulnerabilities have been published in the last few years. The majority of these surveys cover one or two of the following areas: smart contract security vulnerabilities, analysis of existing tools and methodologies to detect smart contract vulnerabilities, formal specification and verification, and common design patterns and practices. Kushwaha et al. [100] conducted a systematic review of Ethereum smart contract analysis tools, classifying them into static and dynamic analysis categories. They scrutinize 86 tools, discussing their methodologies, such as taint analysis, symbolic execution, and fuzzing. However, it lacks consideration for the developer community's preferences and practical scenarios where each analysis tool would be most beneficial. Kushwaha et al. [21] presented a comprehensive systematic review of security vulnerabilities in Ethereum blockchain smart contracts. The paper categorizes vulnerabilities into three main 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. Di Angelo et al. [101] provided a state-of-the-art review of analysis tools of Ethereum smart contracts. The study is based on the actual execution of the tools. It classifies the tools based on availability, maturity level, purpose, and analysis method. However, the survey [101] becomes obsolete (e.g., no longer maintained). Harz et al. [102] presented languages, paradigms, and a verification approach for smart contracts. They did not, however, go into depth about the verification tools or known vulnerabilities. Li et al. [103] discussed blockchain safety problems and proposed enhancements; however, the survey [103] is general because, for example, it included different types of blockchain, such as Ethereum and Bitcoin. Saad et al. [104] investigated several attacks on various blockchain platforms and protection mechanisms. They briefly covered mitigation schemes. However, they did not cover tools to detect vulnerabilities. Atzei et al. [24] presented significant vulnerabilities and attacks related to Ethereum smart contracts. However, they did not cover mitigation schemes. Chen et al. [105] presented 40 types of Ethereum vulnerabilities, their causes, and 29 attacks; however, they did not cover tools to detect vulnerabilities. Zhu et al. [106] presented 11 smart contract vulnerabilities in-depth, along with various defenses against well-known attacks. They covered schemes to detect those vulnerabilities. Durieux et al. [107] conducted an empirical review of nine automated analysis tools on 47,518 contracts; they developed a framework to analyze these tools. The analysis uses two data sets of smart contracts with tagged vulnerabilities. However, the study did not cover vulnerabilities of smart contracts and mitigation schemes.

We conclude that existing surveys focus only on one or two areas of smart contract vulnerabilities. Furthermore, none of the studies discuss the tools based on their output

format.

8. Conclusion

In this study, we presented detailed common vulnerabilities in Ethereum smart contracts and mitigation solutions based on patterns and standards. We have covered the most popular detection methodologies: static analysis, dynamic analysis, and formal verification. We also discussed the benefits and drawbacks of each method. In addition, we provided community-recommended vulnerability detection tools, an execution sample, and detailed feedback for each. Based on our investigation of existing methodologies and tools, we proposed a guideline with the tool(s) for each step in auditing smart contracts.

We observed that most tools detect vulnerabilities but do not provide refactoring recommendations. In future work, we aim to implement a refactoring module that proposes improvements based on the detected bugs and error traces. We also aim to study attacks related to NFTs, such as wash trading, a type of market manipulation in which attackers repeatedly buy and sell the same NFT to increase the price and trading volume.

References

- [1] "Ethereum charts and statistics — etherscan." <https://etherscan.io/stat/supply>, 2022. Accessed: 2023-05-31.
- [2] B. Buterin, "Ethereum white paper," <https://ethereum.org/en/whitepaper/>, 2014.
- [3] S. Team, "Solidity official website ." <https://soliditylang.org/>, 2022. Accessed: 2022-10-02.
- [4] J. Aki, "Blockchain attack: Level finance announces loss of \$1m from smart contract security breach." <https://insidebitcoins.com/news/blockchain-attack-level-finance-announces-loss-of-1m-from-smart-contract-security-breach>. Accessed: 2023-05-10.
- [5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (New York, NY, USA), Association for Computing Machinery, 2016.
- [6] J. Feist, G. Grieco, and A. 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.
- [7] B. 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.
- [8] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1186–1189, 11 2019.
- [9] G. Grieco, W. Song, A. Cygan, J. Feist, and A. 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*, (New York, NY, USA), p. 557–560, Association for Computing Machinery, 2020.

- [10] V. Garousi, M. Felderer, and M. 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), Association for Computing Machinery, 2016.
- [11] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.
- [12] anonymous authors, "Dforce network - rekt." <https://rekt.news/dforce-network-rekt/>. Accessed: 2023-10-05.
- [13] T. 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.
- [14] quadrigainitiative, "Description of events." <https://www.quadrigainitiative.com/casestudy/sirenmarketreentrancybug.php>. Accessed: 2022-04-24.
- [15] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 2–8, 2018.
- [16] openzeppelin, "openzeppelin security." <https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard>. Accessed: 2022-04-20.
- [17] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 910–927, 2020.
- [18] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. 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, (New York, NY, USA), p. 9–16, Association for Computing Machinery, 2018.
- [19] nist, "Cve-2018-10299 detail." <https://nvd.nist.gov/vuln/detail/CVE-2018-10299>. Accessed: 2022-04-20.
- [20] W. Chen, Z. Zheng, E. C.-H. Ngai, P. Zheng, and Y. Zhou, "Exploiting blockchain data to detect smart ponzi schemes on ethereum," *IEEE Access*, vol. 7, pp. 37575–37586, 2019.
- [21] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.
- [22] kingoftheether, "Post-mortem investigation (feb 2016)." <https://www.kingoftheether.com/postmortem.html>. Accessed: 2022-04-20.
- [23] openzeppelin, "openzeppelin the parity wallet hack explained." <https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/>. Accessed: 2022-04-20.
- [24] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust* (M. Maffei and M. Ryan, eds.), (Berlin, Heidelberg), pp. 164–186, Springer Berlin Heidelberg, 2017.
- [25] X. Liu, R. Chen, Y.-W. Chen, and S.-M. Yuan, "Off-chain data fetching architecture for ethereum smart contract," in *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCB)*, pp. 1–4, 2018.
- [26] D. Boneh and M. Naor, "Timed commitments," in *Advances in Cryptology — CRYPTO 2000*, CRYPTO '00, (Berlin, Heidelberg), p. 236–254, Springer-Verlag, 2000.
- [27] randao, "randao github." <https://github.com/randao/randao>. Accessed: 2022-04-20.
- [28] Chainlink, "Chainlink vrf." <https://docs.chain.link/vrf>, 2023. Accessed: 2024-01-02.
- [29] N. Fatima Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 22–29, 2020.
- [30] M. Kaleem, A. Mavridou, and A. 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)*, pp. 107–111, 2020.
- [31] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 259–269, 2018.
- [32] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 01 2018.
- [33] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, (New York, NY, USA), p. 65–68, Association for Computing Machinery, 2018.
- [34] A. López Vivar, A. L. Sandoval Orozco, and L. J. García Villalba, "A security framework for ethereum smart contracts," *Computer Communications*, vol. 172, pp. 119–129, 2021.
- [35] A. Dika and M. 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)*, pp. 955–962, 2018.
- [36] T. Krupa, M. Ries, I. Kotuliak, K. Košťál, and R. Bencel, "Security issues of smart contracts in ethereum platforms," in *2021 28th Conference of Open Innovations Association (FRUCT)*, pp. 208–214, 2021.
- [37] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services, iiWAS2018*, (New York, NY, USA), p. 375–380, Association for Computing Machinery, 2018.
- [38] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," 2021.
- [39] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Bünzli, and M. 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), Association for Computing Machinery, 2018.
- [40] M. Wöhler and U. 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)*, pp. 1513–1520, 2018.
- [41] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18*, (New York, NY, USA), p. 664–676, Association for Computing Machinery, 2018.
- [42] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018.
- [43] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, (New York, NY, USA), p. 454–469, Association for Computing Machinery, 2020.

- [44] swcregistry, “Smart contract weakness classification and test cases.” <https://swcregistry.io/>. Accessed: 2022-04-02.
- [45] securing, “Smart contract security verification standard.” <https://github.com/securing/SCSVS>. Accessed: 2022-04-02.
- [46] W. Wögerer and T. U. Wien, “A survey of static program analysis techniques,” 2005.
- [47] owasp, “Static code analysis.” https://owasp.org/www-community/controls/Static_Code_Analysis. Accessed: 2022-04-02.
- [48] nist, “Opcodes for the evm.” <https://ethereum.org/en/developers/docs/evm/opcodes/>. Accessed: 2023-05-09.
- [49] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [50] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, “Ethersolve: Computing an accurate control-flow graph from ethereum bytecode,” *CoRR*, vol. abs/2103.09113, 2021.
- [51] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, p. 385–394, jul 1976.
- [52] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of model checking*, pp. 305–343, Springer, 2018.
- [53] H. Wang, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, “Oracle-supported dynamic exploit generation for smart contracts,” 2019.
- [54] S. Amani, M. Bégel, M. Bortin, and M. Staples, “Towards verifying ethereum smart contract bytecode in isabelle/hol,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, (New York, NY, USA), p. 66–77, Association for Computing Machinery, 2018.
- [55] D. Annenkov, J. B. Nielsen, and B. 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.
- [56] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “Verx: Safety verification of smart contracts,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1661–1677, 2020.
- [57] ethereum, “Act formal specification.” <https://ethereum.github.io/act/>. Accessed: 2022-04-20.
- [58] “Blockchain security & ethereum smart contract audits.” <https://consensys.net/diligence/>, note = Accessed: 2022-05-12, Year = 2022, author=consensys.
- [59] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, “Ethainter: A smart contract security analyzer for composite vulnerabilities,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 454–469, 2020.
- [60] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, “Kevm: A complete formal semantics of the ethereum virtual machine,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 204–217, 2018.
- [61] L. Mazurek, *EthVer: Formal Verification of Randomized Ethereum Smart Contracts*, pp. 364–380. Springer-Verlag, 09 2021.
- [62] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, (New York, NY, USA), Association for Computing Machinery, 2018.
- [63] J. Ellul and G. J. Pace, “Runtime verification of ethereum smart contracts,” in *2018 14th European Dependable Computing Conference (EDCC)*, pp. 158–163, 2018.
- [64] V. Wüstholtz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (New York, NY, USA), p. 1398–1409, Association for Computing Machinery, 2020.
- [65] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and A. W. Roscoe, “Re-guard: Finding reentrancy bugs in smart contracts,” *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 65–68, 2018.
- [66] I. Grishchenko, M. Maffei, and C. Schneidewind, *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*, pp. 243–269. Springer International Publishing, 04 2018.
- [67] crytic, “slither tool github.” <https://github.com/crytic/slither>. Accessed: 2023-05-31.
- [68] CONSENSYS, “Mythx officil website.” <https://mythx.io/>. Accessed: 2022-03-28.
- [69] ConsenSys, “mythril tool github.” <https://github.com/ConsenSys/mythril>. Accessed: 2023-05-31.
- [70] crytic, “Echidna tool github.” <https://github.com/crytic/echidna>. Accessed: 2023-05-31.
- [71] trailofbits, “manticore tool github.” <https://github.com/trailofbits/manticore>. Accessed: 2023-05-31.
- [72] eth sri, “securify2 tool github.” <https://github.com/eth-sri/securify2>. Accessed: 2023-05-31.
- [73] runtimeverification, “Kevm tool github.” <https://github.com/runtimeverification/evm-semantics>. Accessed: 2023-05-31.
- [74] smartdec, “smartcheck tool github.” <https://github.com/smartdec/smartcheck>. Accessed: 2023-05-31.
- [75] nevillegrech, “Madmax tool github.” <https://github.com/nevillegrech/MadMax>. Accessed: 2023-05-31.
- [76] J. J. Honig, M. H. Everts, and M. Huisman, “Practical mutation testing for smart contracts,” in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pp. 289–303, Springer International Publishing, 2019.
- [77] JoranHonig, “vertigo tool github.” <https://github.com/JoranHonig/vertigo>. Accessed: 2023-05-31.
- [78] SeUniVr, “Ethersolve tool github.” <https://github.com/SeUniVr/EtherSolve>. Accessed: 2022-03-28.
- [79] pventuzelo, “octopus tool github.” <https://github.com/pventuzelo/octopus>. Accessed: 2023-05-31.
- [80] enzymefinance, “Oyente tool github.” <https://github.com/enzymefinance/oyente>. Accessed: 2023-05-2023.
- [81] openzeppelin, “erc20-verifier.” <https://erc20-verifier.openzeppelin.com/>. Accessed: 2022-04-11.
- [82] raineorshine, “solgraph tool github.” <https://github.com/raineorshine/solgraph>. Accessed: 2023-05-31.
- [83] christoftorres, “Osiris tool github.” <https://github.com/christoftorres/Osiris>. Accessed: 2023-05-23.
- [84] “Programming z3.” <http://theory.stanford.edu/~nikolaj/programmingz3.html>, 2022. Accessed: 2022-04-17.
- [85] SmartContractSecurity, “Swc-101 test case.” <https://swcregistry.io/docs/SWC-101#integer-overflow-mapping-sym-1sol>. Accessed: 2022-04-02.
- [86] P. Praitheeshan, L. Pan, and R. Doss, *Security Evaluation of Smart Contract-Based On-chain Ethereum Wallets*, pp. 22–41. Springer-Verlag, 12 2020.
- [87] crytic, “slither list of vulnerabilities.” <https://github.com/crytic/slither#detectors>. Accessed: 2022-04-02.
- [88] crytic, “Slither erc conformance.” <https://github.com/crytic/slither/wiki/ERC-Conformance>. Accessed: 2022-04-20.
- [89] crytic, “Crytic website.” <https://www.crytic.io/>. Accessed: 2022-03-30.
- [90] ConsenSys, “mythril modules.” <https://mythril-classic.readthedocs.io/en/master/module-list.html>. Accessed: 2022-06-11.

- [91] ConsenSys, “Mythx and continuous integration (part 1): Circleci.” <https://blog.mythx.io/howto/mythx-and-continuous-integration-part-1-circleci/>. Accessed: 2022-05-11.
- [92] trailofbits, “Category archives: Manticore.” <https://blog.trailofbits.com/category/manticore/>. Accessed: 2022-04-22.
- [93] trailofbits, “List of ethereum detectors.” <https://github.com/trailofbits/manticore/wiki/Ethereum-Detectors>. Accessed: 2022-06-11.
- [94] “Quickcheck: Automatic testing of haskell programs.” <https://hackage.haskell.org/package/QuickCheck>. Accessed: 2023-05-31.
- [95] trailofbits, “Echidna, a smart fuzzer for ethereum.” <https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/>. Accessed: 2023-05-31.
- [96] 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.
- [97] ethereum, “Smart contract security checklist.” <https://ethereum.org/fr/developers/tutorials/secure-development-workflow/>. Accessed: 2022-04-22.
- [98] vscode, “Solidity visual developer.” <https://marketplace.visualstudio.com/items?itemName=tintinweb.solidity-visual-auditor>. Accessed: 2022-04-22.
- [99] manticore, “Property based symbolic executor: manticore-verifier.” <https://manticore.readthedocs.io/en/latest/verifier.html>. Accessed: 2022-04-22.
- [100] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, “Ethereum smart contract analysis tools: A systematic review,” *IEEE Access*, vol. 10, pp. 57037–57062, 2022.
- [101] M. di Angelo and G. Salzer, “A survey of tools for analyzing ethereum smart contracts,” in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pp. 69–78, 2019.
- [102] D. Harz and W. J. Knottenbelt, “Towards safer smart contracts: A survey of languages and verification methods,” *CoRR*, vol. abs/1809.09805, 2018.
- [103] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, “A survey on the security of blockchain systems,” *Future Generation Computer Systems*, vol. 107, pp. 841–853, 2020.
- [104] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and D. Mohaisen, “Exploring the attack surface of blockchain: A comprehensive survey,” *IEEE Communications Surveys & Tutorials*, vol. PP, pp. 1–1, 03 2020.
- [105] H. Chen, M. Pendleton, L. Njilla, and S. Xu, “A survey on ethereum systems security: Vulnerabilities, attacks, and defenses,” *ACM Comput. Surv.*, vol. 53, no. 3, 2020.
- [106] L.-H. Zhu, B.-K. Zheng, M. Shen, F. Gao, H.-Y. Li, and K.-X. Shi, “Data security and privacy in bitcoin system: A survey,” *Journal of Computer Science and Technology*, vol. 35, pp. 843–862, jul 2018.
- [107] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, “Empirical review of automated analysis tools on 47, 587 ethereum smart contracts,” *CoRR*, vol. abs/1910.10601, 2019.