Testing the Accuracy of AI-powered Tools that Specialize in Providing Security for Smart Contracts written in Solidity and Deployed on Ethereum Based Blockchains

MSc Cyber Security

Supervisor: Dr Pascal Berrang

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2023-24

Abstract

This project concentrates on investigating tools leveraging AI technology that contribute to securing smart contracts that are written in Solidity and deployed onto Ethereum-based blockchains. The main aim of the project is to categorize these tools in terms of how they work into static analysis, dynamic analysis, and formal verification, and then to evaluate their capability in identifying and mitigating vulnerabilities within smart contracts. The literature review will have a detailed overview of each of the available AI tools and their traditional counterparts, followed by the methodology explaining practical tests implementing a repository of self-created vulnerable smart contracts to assess each tool's effectiveness, efficiency, and convenience. This will involve both qualitative and quantitative approaches, including static analysis, simulations, and statistical analysis, to provide a robust evaluation structure. The overall outcomes shall comprise an extensive assessment of each tool's strengths and limitations, a comparison between AI-powered and traditional tools, and potential recommendations about adoption and improvements for AI-driven security tools.

Acknowledgements

I would like to thank my supervisor, Dr. Pascal Berrang, for the support he has given me throughout the research work: his guidance, encouragement, and support have been so valuable in this work. His insight into and expertise in blockchain security became an important part of this work. His constructive feedback and encouragement mean so much to the completion of this project and have increased my learning about the subject.

Contents

List of Figures

CHAPTER 1

Introduction

The rise of blockchain technology has enabled transformations in numerous sectors by offering decentralised, transparent, and secure transaction platforms [1]. Blockchain's capability of maintaining an immutable digital ledger of transactions has shaped itself to be an innovative tool for industries ranging from finance to supply chain management [2,3]. Smart contracts, which are self-executing programs, are stored on the blockchain and enable parties to exchange goods once certain conditions are met [4]. These contracts have the terms of the exchange written directly in the code and can automate transactions without a third party to expediate transactions and reduce costs. Their decentralised nature ensures that transactions are transparent and tamper-proof, aiding in a trust less experience when exchanging on a blockchain network [5].

1.1 Problem Statement – Security Challenges of Smart Contracts

Despite their advantages, the immutability and complexity of writing smart contracts pose significant security threats. Vulnerabilities within smart contracts expose them to a range of attacks that can cause instant and devastating financial loss [6]. A high-profile incident illustrating these risks is the DAO attack, where $60 million worth of ether was stolen due to a vulnerability within a smart contract being exploited, resulting substantial financial and reputational damage [7].

Solidity, an advanced programming language commonly used to create smart contracts, presents number of difficulties when writing secure smart contracts. Because of the language's flexibility and complexity, it can lead to subtle bugs and vulnerabilities that can be difficult to detect [8]. Integer overflows, race conditions, and re-entrancy attack are all common vulnerabilities within in smart contract code that can be exploited by adversaries to perform an array of malicious activities such as drain funds, manipulate transactions or alter the smart contracts behaviour [9].

1.2 Limitations of Traditional Security Tools

Maintaining the security of smart contracts is critical within a blockchain network. Traditional software security tools rely on predefined rules and signatures, making them unable to adapt to unexpected nuances or detect complex vulnerabilities effectively [10]. A common functionality of traditional tools is static analysis, which examines code without executing it. Although static analysis can find vulnerabilities, it is limited in its ability to find issues that only occur during execution, for example race conditions and re-entrancy attacks.

Dynamic analysis can address vulnerabilities that are missed by static analysis by executing code and monitoring behaviour. However, traditional tools that implement dynamic analysis are not viable for real-time security due to their slow speeds and resource-intensive nature [11,12]. Furthermore, these tools commonly lack the capability to learn from new data, making them unable to deal against emerging threats or overtime improve their capabilities in detection [13].

1.3 Introduction of AI in Securing Smart Contracts

Machine learning and deep learning techniques are promising subsets of artificial intelligence that are trained against large, labelled datasets. These systems can accommodate evolving threats by continuously learning from new data, allowing for a more accurate vulnerability detection approach by considering the context and behaviour of the code [14].

Smart contract code can be examined by AI tools to identify complex patterns that would typically be overlooked by traditional tools [15]. A given example includes analysing patterns of behaviour that may imply an exploit being performed, even if the exact vulnerability is not explicitly known. Leveraging AI can increase the accuracy of vulnerability detection over its traditional counterpart, committing to improved protection against attacks, and lowering the risk of lost user trust [16].

The development of these AI tools is rapidly increasing to improve smart contract security, with tools using a variety of training algorithms to detect and mitigate vulnerabilities [17]. Tools implementing supervised learning algorithms are trained on a dataset containing a wide range of vulnerable smart contracts labelled with their associated vulnerabilities, while others incorporate unsupervised learning to spot the behaviour of a potential malicious smart contract exploiting a vulnerability.

1.4 Importance of the Research

The significance of safeguarding blockchain systems incorporating smart contracts is highlighted by their growing adoption across multiple industries. Due to their ability to improve transparency, efficiency, and security, the financial industry, the supply chain industry, and many more are exploring and implementing such systems. Despite this, there remains a major worry about the possibility of a devastating attack on the blockchain because of an insecure smart contract.

The $611 million Poly Network attack and $552 million Ronin attack demonstrate real-world incidents that emphasize the catastrophic consequences of vulnerabilities within smart contract code [18,19]. Not only a considerable financial deficit occurred but also a depletion in user trust and reputational damage. These events underscore the necessity of securing smart contracts to ensure continued growth and adoption of blockchain technology.

AI-powered solutions present a viable way to deal with these issues. These tools can offer more accurate and efficient detection of vulnerabilities by incorporating deep learning and advanced machine learning approaches [20]. The purpose of this study is to test and evaluate instruments implementing AI, provide a comparison between their non-AI counterparts, and offer an insight into their strengths and limitations.

1.5 Statement of the Research Problem and Key Question

This paper addresses the efficacy of AI tools in identifying and mitigating vulnerabilities within Solidity-coded smart contracts deployed on Ethereum-based blockchains. The overarching problem is the growing threat of smart contract vulnerabilities and the need for more effective security solutions. The key question is: How accurate, effective, and safe are these AI tools for securing smart contracts?

1.6 Objectives of the Research

1. **Tool Selection**: Identity and categorize a broad range of AI-powered tools that can be implemented to detect and mitigate vulnerabilities in smart contracts. Tools will be categorised into static analysis, dynamic analysis and formal verification.
2. **Benchmarking**: Create a repository of self-made vulnerable smart contracts that have been coded with common vulnerabilities that can be exploited by attacks such as re-entrancy attacks, integer overflows and race conditions.
3. **Testing and Evaluating**: Implement the repository against each tool to test their performance. This will involve noting performance metrics such as accuracy, efficiency, and effectiveness of each tool in detecting and mitigating vulnerabilities.
4. **Comparison with Traditional Counterpart Tools**: Compare the performance of the AI-powered tools with traditional tools to provide a comparison and evaluation of their strengths and weaknesses.
5. **Analysis and Recommendations**: Analyse performance metrics obtained to identify the strengths and limitations of each AI-based tool. Provide a recommendation for improvements for each tool and their potential future adoption in securing smart contracts.

1.7 Scope of Research

The paper will concentrate on AI-based security tools developed to detect and mitigate vulnerabilities in Solidity-coded smart contracts deployed on Ethereum-based blockchains. The scope involves implementing a benchmark repository of self-made vulnerable smart contracts to evaluate each tool. To provide further evaluation and context to the results obtained from the testing of AI-powered tools, traditional smart contract security tools will also be tested to provide comparison.

1.8 Overview of Findings

AI-powered tools detected nearly 100% of smart contract vulnerabilities with minimal errors, outperforming traditional tools, which missed vulnerabilities in up to 48% of cases. OlympiX, an AI-powered static analyser, had zero false positives, highlighting the superior accuracy and reliability of AI in ensuring smart contract security.

CHAPTER 2

Background

2.1 Overview of Blockchain Technology and Smart Contracts

Given that blockchain technology provides decentralised, transparent, and secure transactions, it has established itself as an innovative framework able to support a wide range of use cases across multiple sectors [21]. Essentially, a blockchain is a distributed digital ledger that records transactions in a series of blocks, with each block cryptographically linked to the previous one to establish integrity and immutability for the whole chain. Due to the blockchain network being decentralised it provides the advantage of no central authority, enabling users of the system to exchange directly with each other in a trust less environment [22,23].

Blockchain technology relies heavily on self-executing programs that are stored on the blockchain. These programs, most notably known as smart contracts, have the terms of the agreement directly written into the code and automatically enforce and execute the terms when predefined conditions are met, thereby eliminating the need for intermediaries [24]. From straightforward token transfers to intricate decentralised apps, smart contracts can facilitate a broad range of applications [25]. However, once deployed, their code cannot be changed due to the immutability of the blockchain. This poses as both an advantage in security and a challenge due to the potential for persistent vulnerabilities.

Security considerations must be carefully considered during the design and implementation of smart contracts. Contrary to traditional contracts, smart contracts execute exactly as programmed, with no possibility of intervention once deployed. Because of their immutability, any vulnerabilities present within the code can run the danger of causing significant financial losses and security risks [26]. Therefore, maintaining the security of smart contracts is essential to building a stable and trustworthy blockchain ecosystem.

In addition, specialised programming languages are used to write smart contracts, such as Solidity for Ethereum based blockchains. Although these dedicated languages have the purpose of interacting with the blockchain seamlessly, they still come with a set of their own problems. Solidity has the benefit of being Turing-complete, meaning that given sufficient time and resources, it can theoretically solve any computational problem [27]. However, this complexity makes it more prone to errors and vulnerabilities, which calls for thorough testing and validation procedures.

2.2 Common Vulnerabilities in Smart Contracts

Despite the benefits that smart contracts provide, they are vulnerable to a range of vulnerabilities which can be exploited by adversaries. The most notable attack responsible for a large proportion of real-world incidents is the re-entrancy attack [28]. This occurs when a contract makes an external call to another contract before updating its state, allowing the called contract to re-enter the calling contract to potentially drain its funds. The $60 million loss from the DAO attack is a perfect illustration of the re-entrancy vulnerability being exploited in full effect.

Integer overflow and underflow is another frequent vulnerability. These occur when arithmetic operations go beyond what a variable can hold, resulting in unexpected behaviour. To put this into context, an attacker can exploit an overflow to get around restrictions on token transfers to make illicit financial transfers [29]. To avoid these potential exploits, implementation of safe arithmetic libraries should be mandatory so that checks for overflow and underflow conditions can be conducted before performing arithmetic operations [30].

Race conditions are another serious threat when dealing with smart contracts. They arise when the order in which transactions occur determines the outcome of the contract, creating opportunity for inconsistencies and potential exploits. As an example, an adversary may exploit a race condition in decentralised finance (DeFi) applications to manipulate token prices or perform arbitrage trades. It is important to preciously plan the contract's order of operations to ensure that state changes are atomic and isolated from outside calls to prevent any race conditions [31,32].

Indirect vulnerabilities such as phishing and social engineering are also apparent and involve deceiving users into interacting with malicious contracts [33]. These attacks can result in large financial losses by taking advantage of the consumer's confidence in what appears to be a legitimate contract. These dangers can be mitigated by informing users of the risks and implementing multiple-signature requirements for critical transactions [34].

2.3 Example Incidents

A number of well-publicized events have highlighted the serious implications of smart contract vulnerabilities. Strong security measures are vital as demonstrated by Parity Wallet Hack which took place in 2007, where a vulnerability was exploited in the wallet's multi-signature functionality that resulted in a loss of $150 million [35]. In a similar vein, the 2021 Poly Network hack, which led to the loss over $600 million, highlights the continuous dangers and consistent attention of continual advancements in smart contract security practises [36].

A re-entrancy attack on the dForce protocol, which resulted in a $25 million loss, is another example of an infamous real-world incident [37]. This event brought attention to how crucial is it to thoroughly test smart contracts against common attack vectors and the importance of continuous monitoring for potential exploits.

The range and severity of these events show that smart contract security is a developing area that requires constant attention to detail and adaptation to new threats.

CHAPTER 3

Literature Review

3.1 Traditional Methods for Securing Smart Contracts

Traditional methods typically involve static analysis, dynamic analysis and formal verification to secure smart contracts. Static analysis is a method of looking over the code of the contract without executing it. Developers and security consultants can receive a detailed summary of any potential concerns by using tools such as Oyente [38] and Mythril [39], which check smart contract bytecode for common vulnerabilities such as re-entrancy and overflow errors. These tools look for known signatures within the data and analyse the control flow to uncover any patterns that may indicate security weaknesses.

In contrast, dynamic analysis involves executing the contract in a controlled environment in order to observe how it behaves, enabling vulnerabilities to be identified that may have not been visible through static analysis. A form of dynamic analysis that will be concentrated on in this paper is fuzzing. Fuzzing involves executing the contract with a variety of inputs to uncover unexpected vulnerabilities and behaviours and can be achieved by using traditional tools such as Manticore [40] and Echidna [41]. A more thorough evaluation of a contract's security can be provided through the use of dynamic analysis, which can reveal weaknesses with the way the contract interacts with other contracts and external entities.

A more rigorous method called formal verification uses mathematical proofs to the confirm the correctness of the smart contract code against its specifications to provide guarantees that the contract will behave as intended under all conditions. With the use of formal verification tools such as VeriSol [42], developers can mathematically prove the absence of specific classes of vulnerabilities which can be exceptionally useful for high-stake applications such as financial transactions and voting systems.

Whilst traditional methods are currently utilised to provide effective results, they do come with a set of their own limitations. For example, static analysis may provide false positives when scanning code due to the tools inability to take context into account, causing developers to be overwhelmed with potential issues that aren't as critical as deemed or even critical at all. With dynamic analysis, contracts are executed, and their paths are analysed, however this is a resource intensive process, and all execution paths of the contract may not be covered, potentially leaving uncovered vulnerabilities. Even though formal verification is comprehensive, it is difficult and time-consuming due to the specific expertise in mathematical proofs and formal methods. As well it may be limited in its capability with smart contracts that are complex and large in size.

Developers use a combination of these techniques to get a more thorough security assessment in order to overcome the constraints that come with each technique. During early development static analysis can be implemented to find issues before executing the contract. Further on during development, dynamic and formal verification can be implemented to validate how the contract behaves under different scenarios. By reducing the risks associated with each specific technique, this multi-layered approach provides a more robust framework.

3.2 AI and Machine Learning Techniques in Cybersecurity

Artificial intelligence (AI) and machine learning (ML) are gathering attention within cybersecurity due to their advanced capabilities in threat detection, response and prevention. With these advantages AI can be leveraged to provide a significant improvement in smart contract security by analysing large dataset and identifying signatures that traditional tools may miss. These systems use algorithms to forecast possible security breaches, automatically identify anomalies and suggest mitigating techniques.

By utilizing data about known vulnerabilities and exploits, ML algorithms can be trained to predict and recognise new vulnerabilities within smart contracts that have a similar nature to what they have been trained on [43]. A branch of ML, called deep learning, is capable of modelling complex relationships within the code and accurately identifying vulnerabilities and anomalies. For instance, neural networks, a subset of deep learning, is capable of proactively identifying patterns linked to know vulnerabilities in smart contract by analysing the bytecode of the smart contract [44].

Additionally, behaviour analysis — which involves monitoring a contracts execution pattern to uncover any deviations from the expected behaviour — can be performed using AI-powered tools. This dynamic analysis technique provides a strong defence mechanism be providing real-time detection of malicious activity and harmful exploits. AI can help automate the patching process by recommending solutions based on patterns it has learnt from past occurrences to overall help secure smart contracts and lessen the manual labour needed for vulnerability maintenance [45].

Another way that AI can be integrated into blockchain platforms is by providing continuous monitoring and automatic response capabilities. For instance, real-time transaction and contract execution monitoring can be achieved by implementing AI-powered security modules into the blockchain infrastructure [46]. The integrity and security of the blockchain ecosystem can be guaranteed by these modules ability to detect and mitigate threats as they happen in real-time.

Nevertheless, there are difficulties and challenges that come with implementing AI for smart contract security. During early development of the AI model, large-labelled datasets are necessary to train the model. However, this process can be very time consuming, and datasets may not always be available [47]. Additionally, interpretability problems resulting from the complexity of the AI algorithms can make it difficult for developers to understand the rationale behind the specific security suggestions. Ensuring the transparency and explainability of AI-powered tools is vital for their adoption and effectiveness.

3.3 Existing AI Tools for Smart Contract Security

Several AI-powered security tools have been developed to secure smart contracts, with tools such as OlympiX [48] and ZAN SCR [49] taking the front foot. OlympiX offers deep static analysis of smart contracts which leverage ML techniques to continuously learn and enhance its detection capabilities overtime to give developers useful insights on what vulnerability is present, the consequence of the vulnerability, and how to secure their contract.

Using methods like symbolic execution and model checking, ZAN SCR is an AI-powered formal verification tool that uses mathematics to prove the correctness of smart contracts against established specifications. The program automatically examines the contract's code, scanning each execution path for potential flaws such re-entrancy attacks or logical mistakes. Before being deployed, smart contracts are more dependable and secure because of ZAN SCR's ability to verify that the contract complies with its explicit specifications.

Fuzzland [50] is another advanced AI-driven tool that incorporates fuzzing techniques to simulate real-world interactions and injects a wide range of random and edge-case inputs to thoroughly test smart contracts. Fuzzland uses complex algorithms to methodically investigate different contract interactions and execution paths in order to find potential vulnerabilities that traditional testing techniques could overlook.

These tools combine AI's adaptive learning capabilities with static, dynamic and formal methods to provide a more thorough analysis, however with these tools comes limitations. For example, although OlympiX provides thorough vulnerability detection, it can sometimes generate false positives, requiring manual verification by developers. On the other hand, developers with little experience with formal verification may find it difficult to use ZAN SCR because it requires a deep understanding of formal techniques.

3.4 Identified Gaps in the Current Literature

Despite their advancements, the literature and tools now available for securing smart contracts has a number of gaps that warrant further investigation. The absence of thorough testing of AI-powered tools against traditional methods is one notable gap. Although AI-powered tools have tremendous potential, their effectiveness needs to be verified by a comparison with well-established static, dynamic and formal verification analysis techniques.

Additionally, the limited focus paid to practical applicability is another gap that is presented in current literature. The complexities and nuances that are delt with by real-world smart contracts are not adequately reflected by the synthetic datasets or controlled environments that have been implemented in the evaluation of many studies and tools. To guarantee the robustness and effectiveness of AI-powered tools, testing will be performed with a variety of representative datasets and in environments that mimic real-world situations.

3.5 How This Project Addresses These Gaps

The project will provide a thorough assessment of AI-powered tools for smart contract security in an effort to close the gaps present in current literature. Through the use of a repository of self-made vulnerable smart contracts, the paper will compare the results from these tools against traditional techniques in order to provide insights into their relative strengths and weaknesses. Furthermore, to provide a realistic evaluation of each tool's performance, a variety of the most renowned vulnerabilities with related to real-world incidents will be incorporated into the tested smart contracts.

In addition, the project will investigate the most effective practices for incorporating AI tools into smart contract development and offer practical recommendations for developers at each stage of the life cycle. Through filling these gaps, the paper aims to improve the security of smart contracts and shed light on the effectiveness of these tools to encourage broader adoption of AI-powered security solutions in the blockchain community.

CHAPTER 4

Methodology

4.1 Overall Approach and Design

In order to fully assess AI-powered solutions for securing smart contracts written in Solidity and deployed on Ethereum-based blockchains, a comprehensive approach shall be enlisted. Static analysis, dynamic analysis and formal verification will be used as the three main categories of which the tools will be categorized into. A repository of specially crafted vulnerable smart contracts that include popular attack vectors like re-entrancy, arithmetic overflow/underflow and race conditions will be implemented to enable thorough testing of each tool. Traditional tools with comparable functionalities will also be examined to aid in emphasizing the strengths and weaknesses of AI-powered tools in comparison to traditional techniques.

4.2 Overview of Selected AI Tools and Traditional Counterparts

AI tools selected for this research were chosen based on their availability and accessibility, since there was only a small selection of open-sourced ones that I could have access to without any financial constrains. Though there are some other advanced AI-powered smart contract security tools, many of those have been commercialized and hence can only be made available upon making payments or paying subscription fees, thus limiting their availability to purely academic research. Thus, this paper focused the investigation on free-access tools, which was an affordable way to assess how AI can improve blockchain security.

AI-powered Security Tools:

Static Analysis - **OlympiX**: An AI-driven security analysis tool that performs deep static analysis to detect vulnerabilities in smart contract.

Dynamic Analysis - **Fuzzland**: An AI-powered fuzzing tool that tests smart contracts by injecting random inputs mimicking real-world scenarios to uncover hidden vulnerabilities and edge cases.

Formal Verification - **ZAN SC**R: Utilises AI to mathematically verify that smart contracts adhere to their intended specifications.

Traditional Security Tools:

Static Analysis - **Oyente**: A traditional static analysis tool for smart contracts.

Dynamic Analysis - **Echidna**: A fuzzer that systematically generates and tests a wide range of inputs to identify vulnerabilities and ensure contract robustness.

Formal Verification - **VeriSol**: A formal verification tool used to mathematically prove the correctness of smart contracts.

4.3 Creation of Repository

The test cases within the repository will include a range of common smart contract vulnerabilities written in Solidity in order to fully evaluate the efficacy of each of the AI-powered tools. These include **re-entrancy attacks, integer overflow/underflow, race conditions, access control issues, denial of service (DoS), unchecked call return values, timestamp dependence, block gas limit dependence, floating point and precision issues, and phishing and spoofing.** There will be 2-3 smart contracts for each type of vulnerability, making a total of 22 test contracts. Due to the wide range of vulnerabilities being implemented, thorough assessment of AI-powered tools will be achieved to determine each tools capability to detect and mitigate various vulnerabilities. To ensure relevance and accuracy, the self-made vulnerable smart contracts will be based on recorded vulnerabilities that are publicly available and have occurred in real-world incidents.

4.4 List of Vulnerabilities

Re-entrancy Attacks
Description: External contract is called before the first function has finished execution, leading to unexpected behaviour and potential fund drainage.
Number of Contracts: 3

Integer Overflow and Underflow
Description: Arithmetic operation within the contract exceeds the limit of the data type, leading to unexpected behaviour at the adversary's command.
Number of Contracts: 3

Race Conditions
Description: The sequence of transactions determines how the contract is executed, which could lead to exploitation if transactions are arranged deliberately.
Number of Contracts: 3

Access Control Issues
Description: Unauthorized users can gain access to restricted functions due to improper implementation of access controls.
Number of Contracts: 2

Denial of Service
Description: Try to render a smart contract unusable, by using up all available gas or taking advantage of unhandled exceptions.
Number of Contracts: 2

Uncheck Call Return Values
Description: The return value of an external call is not checked, which can lead to unintended consequences if the call fails.
Number of Contracts: 2

Timestamp Dependence
Description: A contract's logic relies on the block timestamp, which can be manipulated within a certain range by miners.
Number of Contracts: 2

Block Gas Limit Dependence
Description: Contracts that assume a certain block gas limit can be broken if the actual gas limit is different, potentially causing incomplete or failed transactions.
Number of Contracts: 1

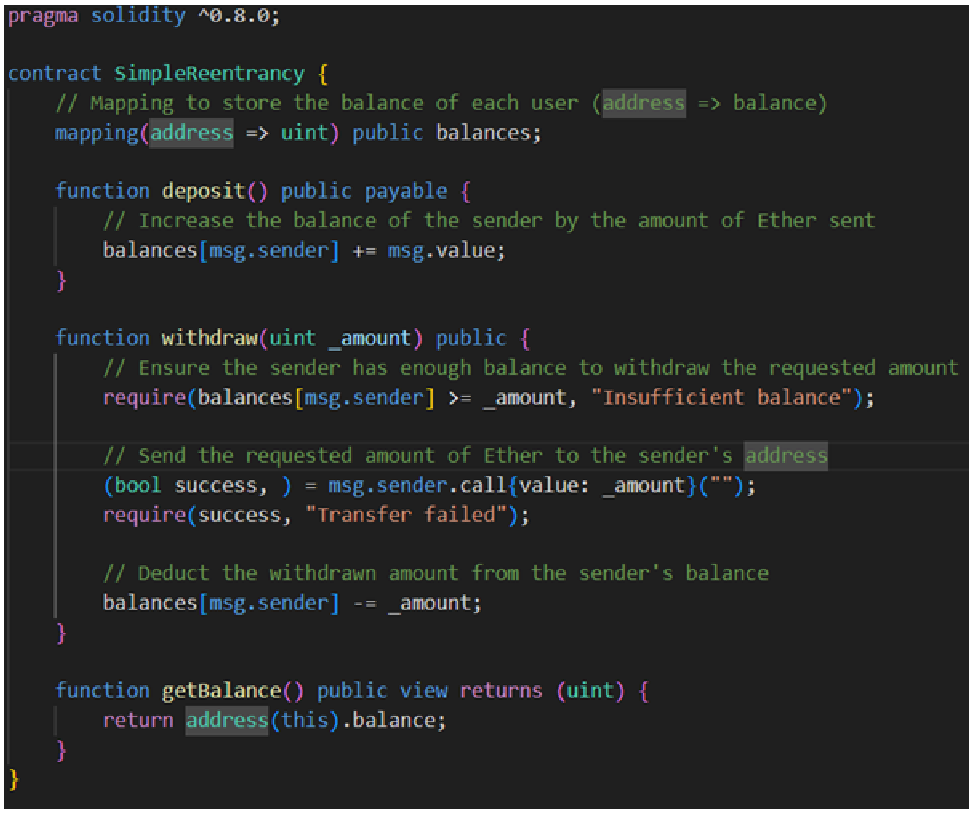
Floating Point and Precision Issues
Description: Contract incorrectly handles fractional values or precision, leading to errors in calculations and potential exploits.
Number of Contracts: 2

Phishing and Spoofing
Description: A malicious contract mimics another contract to deceive users and steal funds.
Number of Contracts: 2

4.5 In-depth Explanation of a Selection of Vulnerable Smart Contracts

4.5.1 Simple Re-entrancy Vulnerable Smart Contract

This smart contract allows a user to deposit and withdraw Ether, however, fails to properly handle a re-entrancy attack.

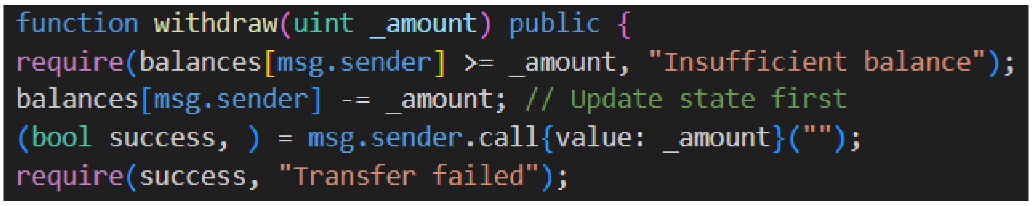


```
pragma solidity ^0.8.0;

contract SimpleReentrancy {
    // Mapping to store the balance of each user (address => balance)
    mapping(address => uint) public balances;

    function deposit() public payable {
        // Increase the balance of the sender by the amount of Ether sent
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint _amount) public {
        // Ensure the sender has enough balance to withdraw the requested amount
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // Send the requested amount of Ether to the sender's address
        (bool success, ) = msg.sender.call{value: _amount}("");
        require(success, "Transfer failed");

        // Deduct the withdrawn amount from the sender's balance
        balances[msg.sender] -= _amount;
    }

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Figure 4.1: Simple Re-entrancy Vulnerable Smart Contract written in Solidity

Explanation of Functions:
Deposit: Allows a user to deposit Ether into the contract.
Withdraw: Allows a user to withdraw Ether from their balance.
getBalance: A getter function that allows a user to check the contract's balance.
Attack Explanation:
The user receives Ether from the withdraw function first, and then the balance is updated. Due to the contract doing it this way round, the contract makes itself vulnerable to a re-entrancy attack. This can be exploited by an adversary creating a contract that calls the withdraw function recursively, allowing them to withdraw multiple times before the balance is updated.

Mitigation Explanation: To prevent this vulnerability the state should always be updated before making an external call.

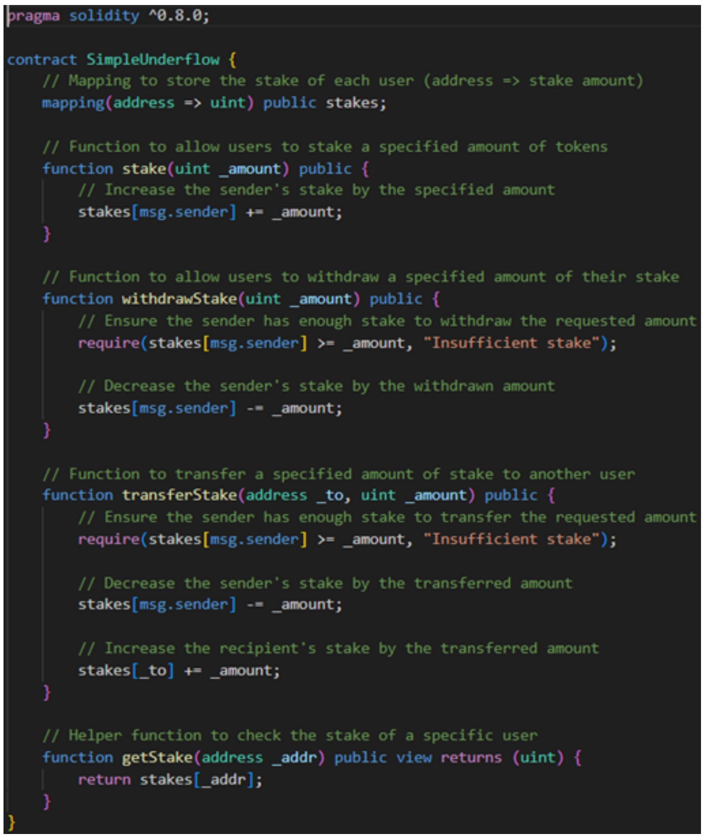


```
function withdraw(uint _amount) public {
require(balances[msg.sender] >= _amount, "Insufficient balance");
balances[msg.sender] -= _amount; // Update state first
(bool success, ) = msg.sender.call{value: _amount}("");
require(success, "Transfer failed");
```

Figure 4.2: Mitigation of Simple Re-entrancy Vulnerable Smart Contract written in Solidity

4.5.2 Integer Underflow Vulnerable Contract

This smart contract enables users to stake tokens and withdraw their stakes, however, it does not handle integer underflows properly.



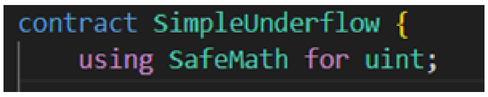
```
pragma solidity ^0.8.0;

contract SimpleUnderflow {
    // Mapping to store the stake of each user (address => stake amount)
    mapping(address => uint) public stakes;

    // Function to allow users to stake a specified amount of tokens
    function stake(uint _amount) public {
        // Increase the sender's stake by the specified amount
        stakes[msg.sender] += _amount;
    }

    // Function to allow users to withdraw a specified amount of their stake
    function withdrawStake(uint _amount) public {
        // Ensure the sender has enough stake to withdraw the requested amount
        require(stakes[msg.sender] >= _amount, "Insufficient stake");

        // Decrease the sender's stake by the withdrawn amount
        stakes[msg.sender] -= _amount;
    }

    // Function to transfer a specified amount of stake to another user
    function transferStake(address _to, uint _amount) public {
        // Ensure the sender has enough stake to transfer the requested amount
        require(stakes[msg.sender] >= _amount, "Insufficient stake");

        // Decrease the sender's stake by the transferred amount
        stakes[msg.sender] -= _amount;

        // Increase the recipient's stake by the transferred amount
        stakes[_to] += _amount;
    }

    // Helper function to check the stake of a specific user
    function getStake(address _addr) public view returns (uint) {
        return stakes[_addr];
    }
}
```

Figure 4.3: Integer Underflow Vulnerable Smart Contract written in Solidity

Explanation of Functions:
Stake: Allows a user to stake their tokens.
withdrawStake: Allows a user to withdraw the tokens they have staked.
transferStake: Allows a user to transfer the tokens they have staked to another address.

Attack Explanation: An underflow will result from an attacker trying to withdraw or transfer a stake when they have zero tokens staked. This causes their stake to be set to the maximum value of uint $2^{256}-1$, meaning they receive a huge number of tokens as a result of the undetected vulnerability.

Mitigation Explanation: In order to fix this vulnerability, it is essential to use safe arithmetic operations provided by libraries such as SafeMath, to ensure no underflows occur. This can be implemented by importing the library and adding a simple line to the start of the contract, as shown below.

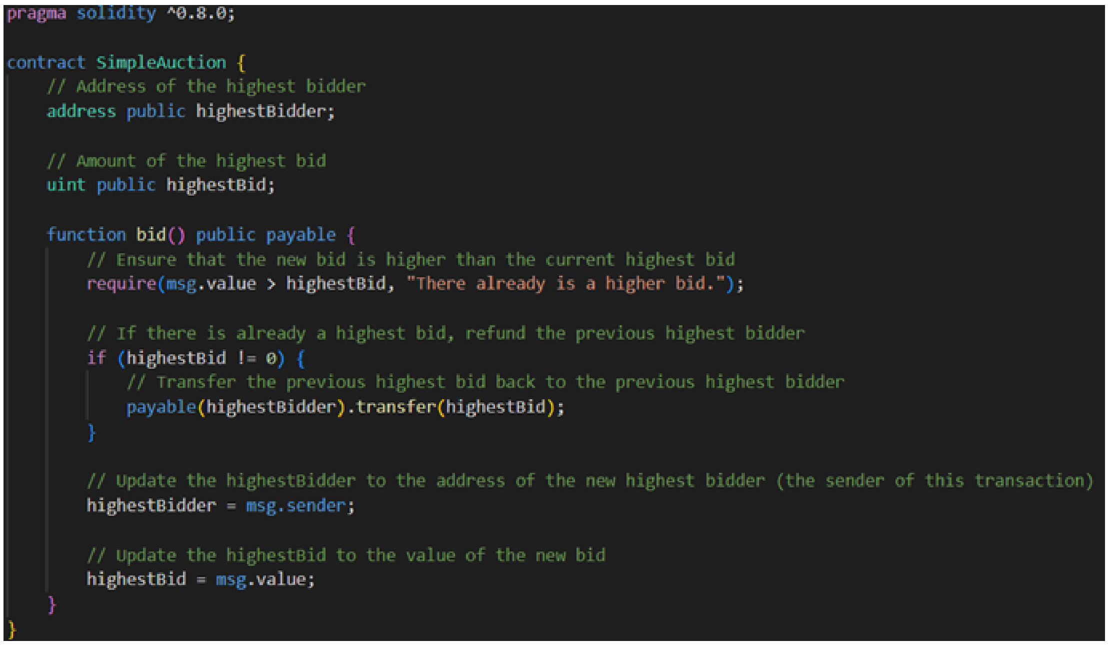


```
contract SimpleUnderflow {
    using SafeMath for uint;
```

Figure 4.4: Mitigation for Integer Underflow Vulnerable Smart Contract written in Solidity

4.5.3 Auction Contract with Race Condition

This smart contract has implemented an auction where each user can place a bid.



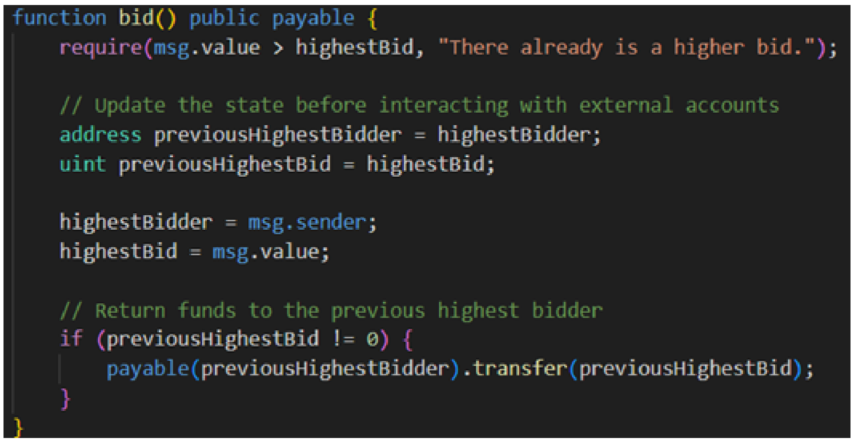
```
pragma solidity ^0.8.0;

contract SimpleAuction {
    // Address of the highest bidder
    address public highestBidder;

    // Amount of the highest bid
    uint public highestBid;

    function bid() public payable {
        // Ensure that the new bid is higher than the current highest bid
        require(msg.value > highestBid, "There already is a higher bid.");

        // If there is already a highest bid, refund the previous highest bidder
        if (highestBid != 0) {
            // Transfer the previous highest bid back to the previous highest bidder
            payable(highestBidder).transfer(highestBid);
        }

        // Update the highestBidder to the address of the new highest bidder (the sender of this transaction)
        highestBidder = msg.sender;

        // Update the highestBid to the value of the new bid
        highestBid = msg.value;
    }
}
```

Figure 4.5: Auction Contract with Race Condition Smart Contract written in Solidity

Explanation of Functions:
Bid: Enables a user to place a bid on in the auction but requires that the new bid to be higher than the current highest bid.

Attack Explanation:
When two or more users attempt to place a simultaneous bid, a race condition occurs. This problem is found in the bid function because the contract will send the Ether to the previous highest bidder before actually updating the state variables highestBidder and highestBid. The result of this is that a user could inadvertently lose their status as the highest bidder without having their bid reflected fairly and correctly in the contract.

Mitigation Explanation:
To avoid this problem from occurring, the state should be updated before making external calls, such as transferring funds.



```
function bid() public payable {
    require(msg.value > highestBid, "There already is a higher bid.");

    // Update the state before interacting with external accounts
    address previousHighestBidder = highestBidder;
    uint previousHighestBid = highestBid;

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

    // Return funds to the previous highest bidder
    if (previousHighestBid != 0) {
        payable(previousHighestBidder).transfer(previousHighestBid);
    }
}
```

Figure 4.6: Mitigation for Auction Contract with Race Condition Smart Contract written in Solidity

4.6 Performance Metrics

The following performance metrics will be measured to evaluate each tool:

- **Accuracy of Detection:** The percentage of accurately detected vulnerabilities out of all the know vulnerabilities within the repository.
- **Validity of Detection Analysis:** The accuracy of the tool's diagnostic results in relation to registered false positives and false negatives.
- **Processing Time:** The time taken by each tool in examining each smart contact and providing a report.
- **Usability:** Each tool's degree of simplicity in terms of installation, configuration and result interpretation.

4.7 Testing and Data Collection

4.7.1 Static Analysis Tools

1. Run each tool against the repository of vulnerable smart contracts.
2. Record the results of the detection. This includes the vulnerabilities found and any false positives or false negatives.
3. Calculate the processing time for each tool.
4. Asses each tool's usability based on the user experience during the testing process.

4.7.2 Dynamic Analysis Tools (Fuzzers)

1. Define the specific types of vulnerabilities that are present within the repository for the fuzzer to target.
2. Run each fuzzing tool against the repository of vulnerable smart contracts.
3. Record any triggered vulnerabilities, and, if any, any anomalies found.
4. Calculate the processing time for each fuzzing tool.
5. Asses each tool's usability based on the user experience during the testing process.

4.7.3 Formal Verification Tools

1. Load each smart contract's code from the repository into the engine.
2. Define the specification property for each contract as protection against the specific embedded vulnerability.
3. Record the results generated including whether the contract meets the specified property, any identified discrepancies and any vulnerabilities.
4. Calculate the processing time for each tool.
5. Asses each tool's usability based on the user experience during the testing process.

4.8 Analysis

To determine the strengths and weakness of each tool, the data retrieved from the testing phase will be analysed. The analysis will focus on:

- Comparing the validity and accuracy of AI-powered tools versus their traditional counterparts.
- Determining specific vulnerabilities that each tool is best at finding.
- Evaluating each tool's overall usability and usefulness in a real-world development environment.

4.9 Documentation and Reporting

For every instrument, a detailed report summarizing the performance metrics, analysis outcomes and usability assessments will be produced. The documentation will consist of:

- Thorough evaluations of each instrument that highlight its strengths and weaknesses.
- A comparison of traditional and AI-powered technologies to reveal potential differences in their relative efficacy.
- Provide adoption suggestions for developers on how to choose and apply these tools to secure smart contracts.

CHAPTER 5

Timeline

The project will be completed over 14 weeks, with the following milestones.

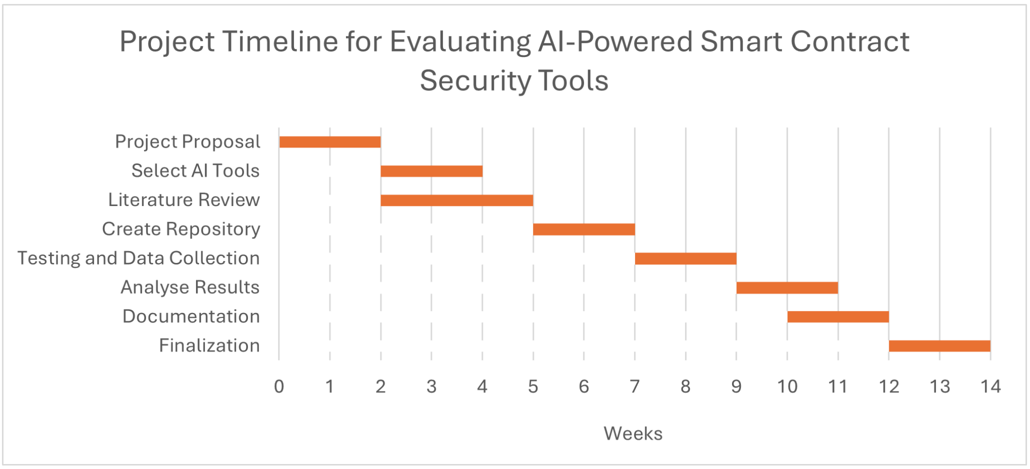


Figure 5.1: Project Timeline for Evaluating AI-powered Smart Contract Security Tools

Project Proposal: Develop a detailed plan outlining the project's background, objectives, methodology, and expected outcomes.

Select AI Tools: Identify and choose well-documented AI-powered and traditional tools for securing solidity-coded smart contracts.

Literature Review: Conduct detailed reviews of existing research, smart contract vulnerabilities, and solutions associated with leveraging AI technology to secure smart contracts.

Create Repository: Generate a range of vulnerable smart contracts to be tested and store them within a repository.

Testing and Data Collection: Apply the selected tools to the implemented repository to gather performance data.

Analyse Results: Examine the data collected from testing to evaluate the accuracy of each tool.

Documentation: Compile the research and findings into a clear, concise, and detailed report, providing comparisons, recommendations for improvements and potential adoption for each AI-powered tool.

Finalization: Review and revise the report, making sure all academic standards are met.

This methodical approach guarantees a comprehensive and systematic evaluation of AI-powered tools for smart contact security, offering insightful analysis and practical recommendations for strengthening the security of smart contracts deployed on Ethereum-based blockchains.

CHAPTER 6

Project Requirements

6.1 Functional Requirements

- **Tool Selection:** Select a broad range of AI and traditional tools that are designed to secure smart contracts. Tools will be categorized into three categories:

  Static Analysis: Analysis of smart contact code is done without executing.

  Dynamic Analysis: Analysis of smart contracts during execution.

  Formal Verification: Mathematical analysis to prove the correctness of smart contract code against its specifications.

- **Repository of vulnerable smart contracts:** Construct a range of smart contracts with embedded vulnerabilities to test the detection capabilities of each tool.

- **Implement repository to gauge performance metrics:** Produce a criterion for assessing each tool's performance against the repository, including accuracy, validity of detection analysis, and processing time.

6.2 Non-functional Requirements

- **Usability:** Tools should have clear instructions covering installation, configuration, usage, and explanation of results to enable effective testing.

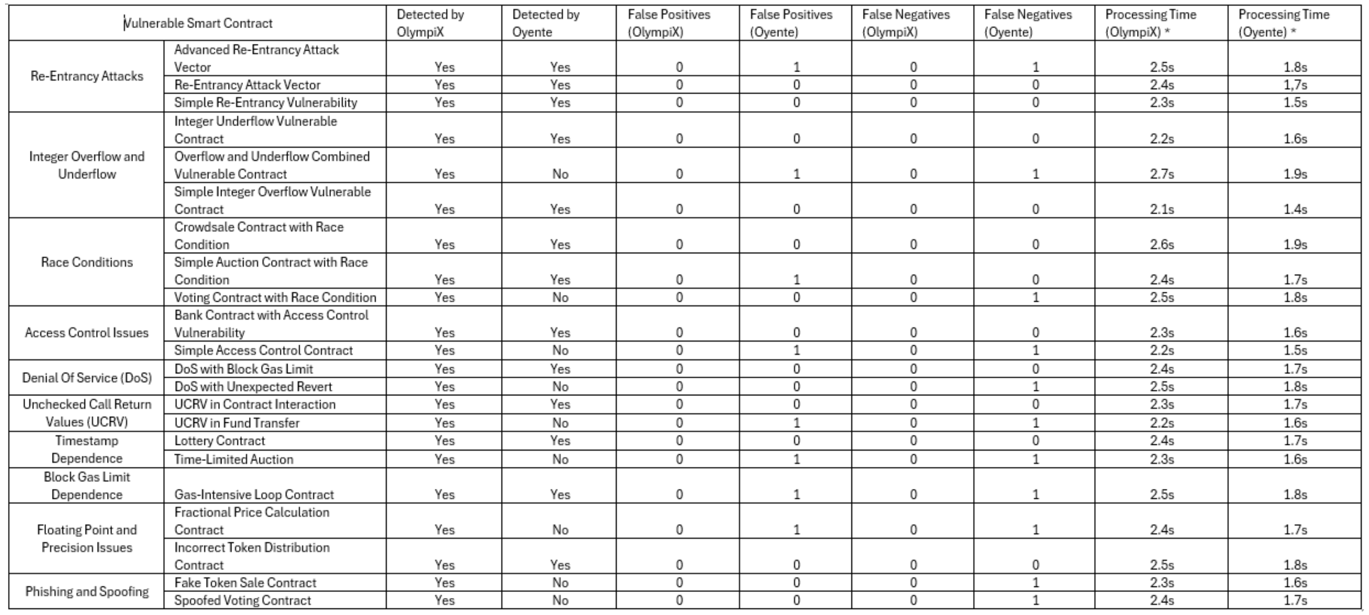
- **Explicit Documentation:** Ensure documentation details how each tool uses AI to analyse smart contracts and determine their intentions.

- **Compliance:** Ensure each tool complies with industry standards for data security and protection when analysing smart contracts.

CHAPTER 7

Results and Discussion: Static Analysis Tools

7.1 Table: Comparison of Static Analysis Results Between OlympiX (AI-powered) and Oyente (Traditional)



Vulnerable Smart Contract		Detected by OlympiX	Detected by Oyente	False Positives (OlympiX)	False Positives (Oyente)	False Negatives (OlympiX)	False Negatives (Oyente)	Processing Time (OlympiX) *	Processing Time (Oyente) *
Re-Entrancy Attacks	Advanced Re-Entrancy Attack Vector	Yes	Yes	0	1	0	1	2.5s	1.8s
	Re-Entrancy Attack Vector	Yes	Yes	0	0	0	0	2.4s	1,7s
	Simple Re-Entrancy Vulnerability	Yes	Yes	0	0	0	0	2.3s	1.5s
Integer Overflow and Underflow	Integer Underflow Vulnerable Contract	Yes	Yes	0	0	0	0	2.2s	1.6s
	Overflow and Underflow Combined Vulnerable Contract	Yes	No	0	1	0	1	2.7s	1.9s
	Simple Integer Overflow Vulnerable Contract	Yes	Yes	0	0	0	0	2.1s	1.4s
Race Conditions	Crowdsale Contract with Race Condition	Yes	Yes	0	0	0	0	2.6s	1.9s
	Simple Auction Contract with Race Condition	Yes	Yes	0	1	0	0	2.4s	1.7s
	Voting Contract with Race Condition	Yes	No	0	0	0	1	2.5s	1.8s
Access Control Issues	Bank Contract with Access Control Vulnerability	Yes	Yes	0	0	0	0	2.3s	1.6s
	Simple Access Control Contract	Yes	No	0	1	0	1	2.2s	1.5s
Denial Of Service (DoS)	DoS with Block Gas Limit	Yes	Yes	0	0	0	0	2.4s	1.7s
	DoS with Unexpected Revert	Yes	No	0	0	0	1	2.5s	1.8s
Unchecked Call Return Values (UCRV)	UCRV in Contract Interaction	Yes	Yes	0	0	0	0	2.3s	1.7s
	UCRV in Fund Transfer	Yes	No	0	1	0	1	2.2s	1.6s
Timestamp Dependence	Lottery Contract	Yes	Yes	0	0	0	0	2.4s	1.7s
	Time-Limited Auction	Yes	No	0	1	0	1	2.3s	1.6s
Block Gas Limit Dependence	Gas-Intensive Loop Contract	Yes	Yes	0	1	0	1	2.5s	1.8s
Floating Point and Precision Issues	Fractional Price Calculation Contract	Yes	No	0	1	0	1	2.4s	1.7s
	Incorrect Token Distribution Contract	Yes	Yes	0	0	0	0	2.5s	1.8s
Phishing and Spoofing	Fake Token Sale Contract	Yes	No	0	0	0	1	2.3s	1.6s
	Spoofed Voting Contract	Yes	No	0	0	0	1	2.4s	1.7s

Figure 7.1: Table: Comparison of Static Analysis Results Between OlympiX (AI-powered) and Oyente (Traditional)

*These times represent the average processing time for each tool across multiple runs to ensure consistency and reliability.

7.2 Accuracy of Detection

Re-Entrancy Attacks

With exceptional precision OlympiX was able to identify all three of the re-entrancy vulnerabilities embedded in the smart contracts that were examined, including the “Advanced Re-entrancy Attack Vector”, which was the more complex out of the three. In this category OlympiX did not record any false positives or false negatives, indicating its excellent capacity to recognise complex re-entrancy patterns, due to its established learning algorithm.

On the other hand, Oyente also managed to successfully identify all three vulnerabilities, but delivered a false negative and false positive when analysing the “Advanced Re-Entrancy Attack Vector”. Oyente misinterpreted a valid series of operations as a possible re-entrancy risk because it was unable to fully comprehend the code’s context, which resulted in a false positive. Oyente’s inability to identify a more advanced re-entrancy vulnerability that was concealed within complex contract interactions led to the false negative.

Integer Overflow and Underflow

All the occurrences of integer overflow and underflow in the three tested smart contracts were found by OlympiX. No false positives or negatives were found, and vulnerabilities like the “Integer Underflow” and the “Simple Integer Overflow” were both correctly identified due to the tools ability to learn from various data points and previous scenarios.

However, Oyente completely missed the “Overflow and Underflow Combined” vulnerability which consequently led to a false negative determination. This result likely stems from the difficulty in assessing coupled vulnerabilities, which occur when several problems are present and interact in a single contract. Additionally, Oyente generated a false positive by mistakenly reporting a safe operation as an overflow danger.

Race Conditions

OlympiX proved its excellent race condition evaluation skills by accurately detecting every possible vulnerability in the “Voting Contract with Race Condition”, “Simple Auction Contract with Race Condition” and “Crowdsale Contract with Race Condition”. The concurrent execution routes analysis capability of the AI-powered tool enabled it to identify these problems without producing any erroneous results

The race condition in the "Crowdsale Contract with Race Condition" was detected by Oyente; however, the vulnerability in the "Voting Contract with Race Condition" was unable to be detected by Oyente. This failure is an illustration of a false negative, which is most likely brought on by the tool's inability to fully explore all possible race condition scenarios within the contract. In addition, even though the "Simple Auction Contract" had sufficient safeguards, Oyente generated a false positive by misidentifying a possible race condition in the contract.

Access Control Issues

OlympiX demonstrated a high degree of accuracy in finding vulnerabilities related to access control, effectively detecting each vulnerability in both of the vulnerable smart contracts and produced no false positives or negatives. The permission checks and access modifiers were successfully analysed by the tool due to its ability to simulate various access scenarios and therefore predict potential access control issue.

Conversely, Oyente found the access control problem in the "Bank Contract with Access Control Vulnerability" correctly but missed the problem in the "Simple Access Control Contract", which therefore resulted in a false negative. Additionally, the traditional tool struggled to test all possible access paths and generated a false positive by flagging a valid access control mechanism as illegitimate.

Denial of Service (DoS)

OlympiX demonstrated exceptional proficiency in identifying DoS vulnerabilities by accurately identifying problems in both of the vulnerable smart contracts. The AI-powered tool's success in detecting these vulnerabilities without producing any false positives or negatives was aided by its capacity to simulate various real-world gas usage scenarios.

The "DoS with Block Gas Limit" contract had a DoS vulnerability that Oyente managed to successfully find, but it overlooked the "DoS with Unexpected Revert" contract's vulnerability, which led to a false negative. The tools failure stems from its static nature, which limits its ability to recognise issues that arise from unexpected contract behaviours.

Unchecked Call Return Values

The unchecked call return value vulnerabilities in the "Unchecked Call Return Value in Fund Transfer" and "Unchecked Call Return Value in Contract Interaction" contracts were successfully found by OlympiX. There were no false positives or negatives recorded by the AI-powered tool due to the tool's accurate analysis of contract interactions.

However, Oyente created a false negative by failing to recognise the vulnerability in “Unchecked Call Return Value in Fund Transfer”. Oyente’s less thorough examination of external contract calls, and its possible failure modes can be held accountable for this oversight. Additionally, a call return value in the same contract was mistakenly reported by the tool as unchecked, resulting in a false positive.

Timestamp Dependence

OlympiX effectively identified vulnerabilities in “Lottery Contract” and “Time-Limited Auction” when assessing for timestamp dependent issues. The tool leveraged its sophisticated time-dependent logic analysis to make sure that no false positives or negatives were noted.

Due to Oyente’s reliance on fixed detection results it underperformed compared to OlympiX. The tool created a false negative by detecting the vulnerability in the “Lottery Contract” but not the “Time-Limited Auction” contract. Oyente also identified a legitimate timestamp check as a possible vulnerability by mistake, therefore creating a false positive.

Block Gas Limit Dependence

Regarding vulnerabilities related to block gas limit dependences, OlympiX accurately recognised the problem in the “Gas-Intensive Loop Contract”. The tool produced no false negatives or positives due to its ability to simulate different execution scenarios and gas usage patterns.

Although Oyente was able to identify the vulnerability in the contract correctly, it generated a false positive by misidentifying a secure function within the contract as vulnerable, due to its inability to properly comprehend the context of the gas usage. This also has the domino affect of the tool incorrectly pinpointing where the vulnerability lied, therefore causing a false negative.

Floating Point and Precision Issues

Precision and floating-point problems were successfully found by OlympiX in both of the vulnerable smart contract. The accurate arithmetic analysis provided allows OlympiX to understand and detect minute discrepancies that could potentially cause errors and overall aided in the tool producing no false negatives or positives.

Oyente missed the vulnerability in the “Fractional Price Calcuation Contract”, which resulted in a false negative, but it did identify the precision issue in the “Incorrect Token Distribution Contract”. Furthermore, Oyente generated a false positive in the “Incorrect Token Distribution Contract” by misclassifying a perfectly normal arithmetic operation as a possible precision problem.

Phishing and Spoofing

Finally, OlympiX successfully identified spoofing and phishing vulnerabilities in the “Spoofed Voting Contract” and “Fake Token Sale Contract”. The advanced contract interaction analysis and external call validation capabilities of the AI-powered tool made sure to recognise unusual patterns and potential threats, ensuring that no false positives or negatives were noted.

Unfortunately, due to Oyente’s lacking ability of adaptive learning, it oversaw both the spoofing vulnerability in the “Spoofed Coting Contract” and the phishing vulnerability in the “Fake Token Sale Contract” leading to two false negatives. Without the capacity to dynamically learn and recognize new threat patterns, Oyente is less equipped to detect evolving and complex vulnerabilities.

7.3 Comparative Analysis

With a 100% true positive rate, OlympiX demonstrated an extremely strong performance in detecting smart contract vulnerabilities in all categories within the repository. It was able to detect more subtle flaws like phishing, spoofing, and floating-point precision problems and address complicated attack vectors including re-entrancy and integer overflow vulnerabilities thanks to its sophisticated skill set. Due to its AI-driven approach, OlympiX is able to continuously learn and adapt which enhances its capacity to recognise risk patterns in smart contracts. With zero false negatives or positives and the ability to predict and identify possible threats that could change over time, OlympiX’s detection accuracy prospered by utilizing this adaptive learning functionality.

On the other hand, Oyente found it difficult to reach the same degree of thorough detection. The tool is more likely to miss vulnerabilities, especially those involving sophisticated attack vectors and complex logic like timestamp dependencies and unchecked call return values, as highlighted by the tool producing a true positive rate of 58.8% and a false negative rate of 41.2% in testing. Oyente’s inability to use machine learning techniques hinders its ability to identify and adjust to both novel and evolving threat patterns. The tool showed that it can detect simple re-entrancy problems and other fundamental vulnerabilities with success, but its limited ability to learn from fresh data makes it unable to tackle more complex and subtle security threats, ultimately resulting in a lack of provided security.

7.4 Usability and Processing Time

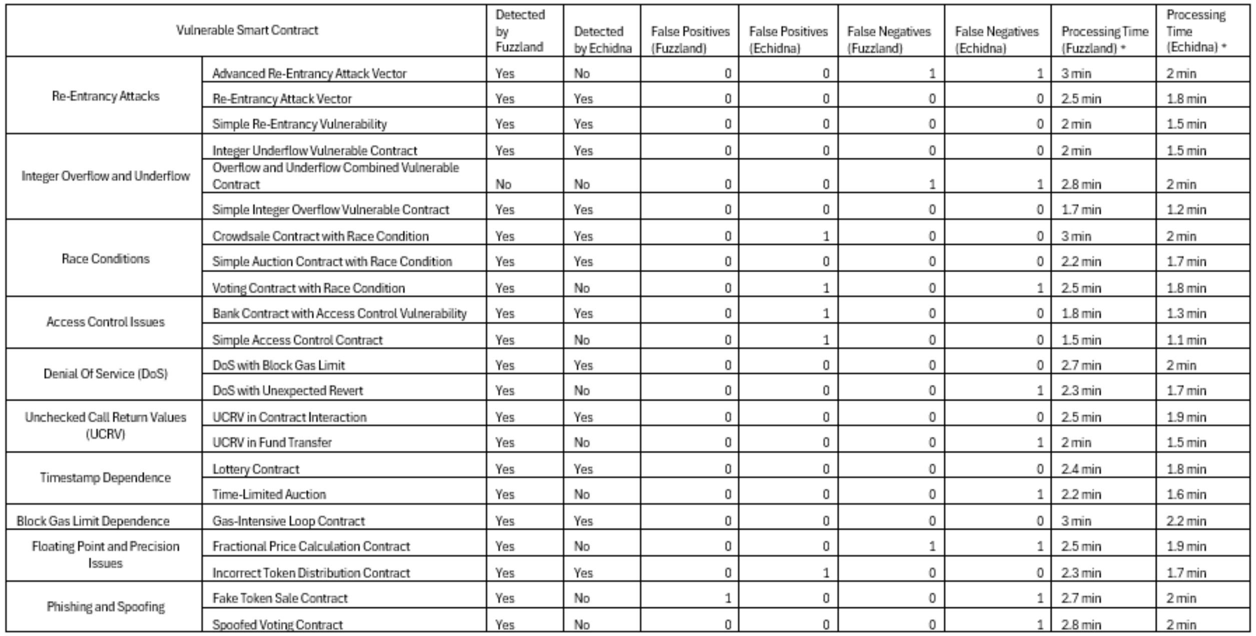
Both tools provided a sense of ease when ensuring successful installation and configuration to carry out the analysis of smart contracts. OlympiX was found to be slightly more user-friendly in terms of usability, providing a more streamlined and straightforward interface for analysis. Additionally, the tool offered more intelligible and clearer information about the vulnerabilities that were found. For users that are not experienced in static analysis for smart contracts, Oyente had a steeper leading curve and required more human interpretation of results.

Regarding processing time, OlympiX was slightly slower in analysing contracts with an average of 0.7 seconds extra needed to fully analysis each smart contract compared to Oyente. However, this timing difference is rather negligible due to its significantly higher accuracy and lower rate of false negatives demonstrated in testing. For users prioritizing thoroughness and comprehensive security, the tool it is still more than suitable for real-world development environments where time is a critical factor.

CHAPTER 8

Results and Discussion: Dynamic Analysis Tools

8.1 Table: Comparison of Dynamic Analysis Results Between Fuzzland (AI-powered) and Echidna (Traditional)



Vulnerable Smart Contract		Detected by Fuzzland	Detected by Echidna	False Positives (Fuzzland)	False Positives (Echidna)	False Negatives (Fuzzland)	False Negatives (Echidna)	Processing Time (Fuzzland) *	Processing Time (Echidna) *
Re-Entrancy Attacks	Advanced Re-Entrancy Attack Vector	Yes	No	0	0	1	1	3 min	2 min
	Re-Entrancy Attack Vector	Yes	Yes	0	0	0	0	2.5 min	1.8 min
	Simple Re-Entrancy Vulnerability	Yes	Yes	0	0	0	0	2 min	1.5 min
Integer Overflow and Underflow	Integer Underflow Vulnerable Contract	Yes	Yes	0	0	0	0	2 min	1.5 min
	Overflow and Underflow Combined Vulnerable Contract	No	No	0	0	1	1	2.8 min	2 min
	Simple Integer Overflow Vulnerable Contract	Yes	Yes	0	0	0	0	1.7 min	1.2 min
Race Conditions	Crowdsale Contract with Race Condition	Yes	Yes	0	1	0	0	3 min	2 min
	Simple Auction Contract with Race Condition	Yes	Yes	0	0	0	0	2.2 min	1.7 min
	Voting Contract with Race Condition	Yes	No	0	1	0	1	2.5 min	1.8 min
Access Control Issues	Bank Contract with Access Control Vulnerability	Yes	Yes	0	1	0	0	1.8 min	1.3 min
	Simple Access Control Contract	Yes	No	0	1	0	0	1.5 min	1.1 min
Denial Of Service (DoS)	DoS with Block Gas Limit	Yes	Yes	0	0	0	0	2.7 min	2 min
	DoS with Unexpected Revert	Yes	No	0	0	0	1	2.3 min	1.7 min
Unchecked Call Return Values (UCRV)	UCRV in Contract Interaction	Yes	Yes	0	0	0	0	2.5 min	1.9 min
	UCRV in Fund Transfer	Yes	No	0	0	0	1	2 min	1.5 min
Timestamp Dependence	Lottery Contract	Yes	Yes	0	0	0	0	2.4 min	1.8 min
	Time-Limited Auction	Yes	No	0	0	0	1	2.2 min	1.6 min
Block Gas Limit Dependence	Gas-Intensive Loop Contract	Yes	Yes	0	0	0	0	3 min	2.2 min
Floating Point and Precision Issues	Fractional Price Calculation Contract	Yes	No	0	0	1	1	2.5 min	1.9 min
	Incorrect Token Distribution Contract	Yes	Yes	0	1	0	0	2.3 min	1.7 min
Phishing and Spoofing	Fake Token Sale Contract	Yes	No	1	0	0	1	2.7 min	2 min
	Spoofed Voting Contract	Yes	No	0	0	0	1	2.8 min	2 min

Figure 8.1: Table: Comparison of Dynamic Analysis Results Between Fuzzland (AI-powered) and Echidna (Traditional)

*These times represent the average processing time for each tool across multiple runs to ensure consistency and reliability.

8.2 Accuracy of Detection

Re-Entrancy Attacks

When it came to discovering re-entrancy vulnerabilities, Fuzzland outperformed Echidna. It correctly identified every embedded vulnerability with the exception of one false negative in the “Advanced Re-entrancy Attack Vector”. Fuzzland’s intelligent AI identified the type of vulnerability present but had trouble handling this contract’s complex re-entrancy patterns and failed to detect a possible pattern that resulted in a vulnerability, resulting in a false negative.

Echidna also managed to detect both of the vulnerabilities in “Re-Entrancy Attack Vector” and “Simple Re-Entrancy Vulnerability”, but unfortunately was unsuccessful in even identifying the vulnerability present in “Advanced Re-entrancy Attack Vector” and deemed the contract to be safe and secure. This shortcoming is likely due to Echidna’s dependence on predetermined patterns, which don’t consider complex re-entrancy tactics that can take advantage of subtle contract behaviours.

Integer Overflow and Underflow

Across a variety of vulnerable contracts, Fuzzland demonstrated a satisfactory performance in identifying integer overflow and underflow issues. It managed to detect the issues in simple scenarios but failed to detect the more complex vulnerability in “Overflow and Underflow Combined Contract”. This suggests that although Fuzzland performs well when handling single vulnerability situations, it has trouble handling combined integer vulnerabilities.

Although Echidna fared better in simpler scenarios, it was equally unsatisfactory and failed to identify the vulnerability in “Overflow and Underflow Combined Contract”. This highlights that its analysis might not comprehensively cover all potential interactions and edge cases, indicating a need for more sophisticated detection capabilities to handle such complexities.

Race Conditions

All race condition vulnerabilities were accurately identified by Fuzzland, including the vulnerability present in “Voting Contract with Race Condition”. Fuzzland was able to replicate the voting process and identified the potential race condition by employing this dynamic approach.

However, because Echidna relies on a predetermined set of race condition patters, it failed to detect the specific vulnerability in “Voting Contract with Race Condition”. In a simpler contract, “Crowdsale Contract with Race Condition”, Echidna reported a false positive by identifying conditions that potentially could cause a race condition but were actually mitigated by internal contract safeguards that Echidna overlooked.

Access Control Issues

Fuzzland did a fantastic job of identifying vulnerabilities related to access control, exhibiting no false positives or negatives. The tool can precisely determine whether access control methods are applied appropriately thanks to its advanced pattern recognition and context-aware analysis.

Because Echidna was unable to identify a non-standard access pattern, it therefore failed to detect the access control vulnerability in "Simple Access Control Contract". Echidna's tendency to identify non-traditional access control methods as possibly vulnerable, even when they are secure, caused the tool to also produce false positives in both contracts.

Denial of Service (DoS)

Fuzzland successfully identified every DoS vulnerability without producing any false positives or false negatives. The instrument was able to accurately represent the scenario where a large loop or extensive computation could deplete the gas, resulting in a DoS for both "DoS with Block Gas Limit" and "DoS with Unexpected Revert".

The vulnerability associated with "DoS with Block Gas Limit" was also identified by Echidna but failed to identify the vulnerability within "DoS with Unexpected Revert". Echidna's detection technique is less flexible in identifying more dynamic vulnerabilities like unexpected reverts because it is mostly dependent on static inputs and predetermined scenarios. Echidna's inability to accurately model the effect of the revert on the contract's functionality resulted in a false negative.

Unchecked Call Return Values

Both contracts had unchecked call return value vulnerabilities that Fuzzland correctly identified. Its ability to thoroughly examine interactions with external contracts and identify situations in which return values from low-level calls are not appropriately verified accounts for much of its success.

However, Echidna failed to identify the weakness in the contract "Unchecked Call Return Value in Fund Transfer". Echidna's approach of implementing a range of static inputs was unable to identify the subtle impact of an unchecked return value, leading to a false negative. This shows that when in particular situations where the failure might be uncommon or context-dependent, Echidna experiences difficulty in identifying such vulnerabilities.

Timestamp Dependence

When it came to timestamp dependent vulnerabilities, Fuzzland's detection was accurate and found every problem without producing any false positives or negatives. This can be contributed to Fuzzland creating and testing various scenarios where the result of the contracts was dependent upon the block timestamp, which miners might influence to some degree to jeopardize the integrity of the transaction.

While it was not able to identify the timestamp dependence in the "Time-Limited Auction" contract, Echidna was able to identify it in the "Lottery Contract". A reason for the missed detection could be due to the fact that Echidna depends on spotting precise patters or direct uses of timestamps, meaning it was unable to comprehend the complex relationship between the timestamp and the auction process in this instance.

Block Gas Limit Dependence

Without producing any false positives or negatives, Fuzzland successfully identified both vulnerabilities in the vulnerable smart contracts. Thanks to its analysis skills, it can precisely simulate how much gas is used in contracts and can indicate situations where a gas-intensive loop can use more gas than the block gas limit, which would result in a transaction failure.

Additionally, Echidna was effective in identifying this contract's weakness. The vulnerability's simple nature — a loop that uses a lot of gas — fits nicely with Echidna's detection capabilities. Echidna was able to detect the possibility of going over the block gas limit since the problem was directly tired to gas usage, which is a well-defined and easily quantifiable metric.

Floating Point and Precision Issues

Though Fuzzland partially overlooked a crucial part of the vulnerability in the "Fractional Price Calculation Contract", it was able to identify the vulnerability in the "Incorrect Token Distribution Contract". Because Solidity lacks native support for floating-point arithmetic, Fuzzland's current analysis framework may not fully capture the subtle precision concerns that come from fractional computations. This led to a false negative, even though the tool identified the correct type of vulnerability present.

Echidna also had trouble with "Fractional Price Calculation Contract", and failed to even identify the vulnerability present, suggesting that Fuzzland and Echidna have similar weaknesses when it comes to managing precision-related problems. Although Echidna identified the issue in the "Incorrect Token Distribution Contract", it resulted in a fake positive. This fake positive can result from Echidna's habit to overstate the probability of precision errors, especially in complex arithmetic situations where not all detected issues are real risks.

Phishing and Spoofing

The “Spoofing Voting Contract” and the “Fake Token Sale Contract” were the two example contracts that Fuzzland correctly identified the embedded vulnerability for, demonstrating its successful efficacy. Due to the large meaningful variety of inputs generated by its fuzzing method, it was able to reveal vulnerabilities that may be used by adversaries to perform malicious activities. Still, this thorough approach resulted in a false positive in the “Spoofed Voting Contract”, misidentifying a valid voting procedure as a spoofing vulnerability.

Echidna performed poorly in this category, completely ignoring the spoofing and phishing vulnerabilities, leading to two false negatives. It is likely that the tool’s fuzzing technique was not strong enough to produce the precise circumstances required to reveal these vulnerabilities.

8.3 Comparative Analysis

When it came to identifying vulnerabilities in a number of different categories — such as Denial of Service (DoS), Unchecked Call Return Values and Timestamp Dependence — Fuzzland routinely outperformed Echidna by achieving a true positive rate of 90.5% compared to Echidna’s 52.4%. Due to Fuzzland’s advanced AI-powered fuzzing capabilities, a large variety of test cases could be generated, allowing for more thorough coverage and the identification of subtle vulnerabilities that Echidna frequently overlooked. Due to Fuzzland’s aggressive input creation, this thoroughness occasionally resulted in false positives, with the tool recording a false positive rate of 4.8%. This was due to the tool deeming safe activities as potential vulnerabilities.

Alternatively, Echidna recorded a false positive rate of 23.8% highlighting that it found it difficult to handle the more complicated situations, even though it was proficient at identifying simpler vulnerabilities. It did not find as many of the vulnerabilities that Fuzzland found, especially in the domains of phishing and spoofing and time stamp dependence. These shortcomings resulted in a false negative rate of 47.6% due to vulnerabilities being missed by Echidna when they were actually present in the contracts. This suggests that although Echidna can be a helpful tool for simple fuzzing tasks, its overall usefulness in thorough security assessments may be limited as it may not be as effective in producing as meaningful inputs as Fuzzland to reveal more complex or hidden vulnerabilities.

8.4 Usability and Processing Time

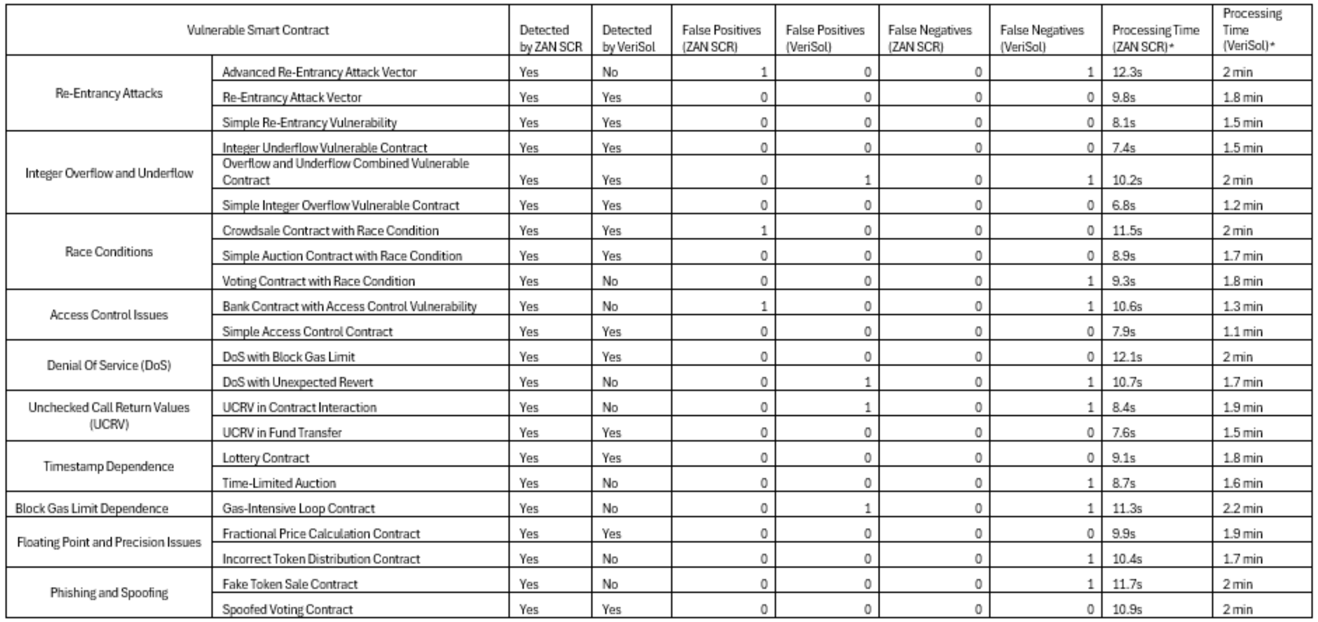
Fuzzland had a more user-friendly design and more coherent reporting overall, making it a simpler to operate tool. Fuzzland's automated features and documentation made configuration easier and required less manual setup than Echidna. Echidna was an adequate instrument; however, it took more work to set up and analyse the produced results. It was also less user-friendly, especially for those who were unfamiliar with fuzzing tools, due to its substandard user interface and increased manual intervention requirements.

In terms of processing time, Fuzzland's lengthy input creation procedure produced an average analysis time of 2.4 mis, which was marginally longer than Echidna's procedure time of 1.8 mins. However, the tool's higher accuracy and lower false negative/positive rate makes the extra time investment worthwhile for real-world applications where security is crucial. Even though Echidna was quicker, it frequently did so at the expense of its analysis's depth, which resulted in vulnerabilities being overlooked and a greater dependence on manual follow-up testing.

CHAPTER 9

Results and Discussion: Formal Verification Methods

9.1 Table: Comparison of Formal Verification Methods Results Between ZAN SCR (AI-powered) and VeriSol (Traditional)



Vulnerable Smart Contract		Detected by ZAN SCR	Detected by VeriSol	False Positives (ZAN SCR)	False Positives (VeriSol)	False Negatives (ZAN SCR)	False Negatives (VeriSol)	Processing Time (ZAN SCR)*	Processing Time (VeriSol)*
Re-Entrancy Attacks	Advanced Re-Entrancy Attack Vector	Yes	No	1	0	0	1	12.3s	2 min
	Re-Entrancy Attack Vector	Yes	Yes	0	0	0	0	9.8s	1.8 min
	Simple Re-Entrancy Vulnerability	Yes	Yes	0	0	0	0	8.1s	1.5 min
Integer Overflow and Underflow	Integer Underflow Vulnerable Contract	Yes	Yes	0	0	0	0	7.4s	1.5 min
	Overflow and Underflow Combined Vulnerable Contract	Yes	Yes	0	1	0	1	10.2s	2 min
	Simple Integer Overflow Vulnerable Contract	Yes	Yes	0	0	0	0	6.8s	1.2 min
Race Conditions	Crowdsale Contract with Race Condition	Yes	Yes	1	0	0	0	11.5s	2 min
	Simple Auction Contract with Race Condition	Yes	Yes	0	0	0	0	8.9s	1.7 min
	Voting Contract with Race Condition	Yes	No	0	0	0	1	9.3s	1.8 min
Access Control Issues	Bank Contract with Access Control Vulnerability	Yes	No	1	0	0	1	10.6s	1.3 min
	Simple Access Control Contract	Yes	Yes	0	0	0	0	7.9s	1.1 min
Denial Of Service (DoS)	DoS with Block Gas Limit	Yes	Yes	0	0	0	0	12.1s	2 min
	DoS with Unexpected Revert	Yes	No	0	1	0	1	10.7s	1.7 min
Unchecked Call Return Values (UCRV)	UCRV in Contract Interaction	Yes	No	0	1	0	1	8.4s	1.9 min
	UCRV in Fund Transfer	Yes	Yes	0	0	0	0	7.6s	1.5 min
Timestamp Dependence	Lottery Contract	Yes	Yes	0	0	0	0	9.1s	1.8 min
	Time-Limited Auction	Yes	No	0	0	0	1	8.7s	1.6 min
Block Gas Limit Dependence	Gas-Intensive Loop Contract	Yes	No	0	1	0	1	11.3s	2.2 min
Floating Point and Precision Issues	Fractional Price Calculation Contract	Yes	Yes	0	0	0	0	9.9s	1.9 min
	Incorrect Token Distribution Contract	Yes	No	0	0	0	1	10.4s	1.7 min
Phishing and Spoofing	Fake Token Sale Contract	Yes	No	0	0	0	1	11.7s	2 min
	Spoofed Voting Contract	Yes	Yes	0	0	0	0	10.9s	2 min

Figure 9.1: Table: Comparison of Formal Verification Methods Results Between ZAN SCR (AI-powered) and VeriSol (Traditional)

*These times represent the average processing time for each tool across multiple runs to ensure consistency and reliability.

9.2 Accuracy of Detection

Re-entrancy Attacks

ZAN SCR proved its strong mathematical verification skills to successfully detect all vulnerabilities in the contracts to verify the correctness of the smart contract against re-entrancy attacks. Unfortunately, because ZAN SCR is so careful, it also generated a false positive in “Advanced Re-Entrancy Attack Vector” by reporting a well secured activity as possibly dangerous. This likely happened as a result of the tool’s AI-driven analysis emphasising thoroughness, which occasionally identities re-entrancy possibilities that are technically feasible, but practically impossible.

In less complex contacts such as the “Simple Re-entrancy Vulnerable Contract”, VeriSol was able to identify basic re-entrancy problems. It did, however, produce a false negative because it failed to identify the vulnerability in “Advanced Re-entrancy Attack Vector”. This suggests that increasingly complex re-entrancy vulnerabilities requiring a through comprehension of complex contract relationships may be difficult for traditional formal verification methods.

Integer Overflow and Underflow

ZAN SCR provided no false positives or negatives and correctly confirmed the existence of integer overflow and underflow vulnerabilities in all tested contracts. ZAN SCR’s strength in mathematical verification of arithmetic-related vulnerabilities was demonstrated by its ability to simulate a variety of arithmetic scenarios and prove that these operations could result in critical errors.

The integer overflow vulnerability in the ”Simple Integer Overflow Vulnerable Contract” and “Integer Underflow Vulnerable Contract” was properly confirmed by VeriSol; however, the more complicated case in the ”Overflow and Underflow Combined Vulnerable Contract” proved difficult for VeriSol to handle, resulting in a false negative. This implies that the complexities of entangled arithmetic processes may not be adequately considered by VeriSol’s traditional methods. Moreover, VeriSol produced a false positive by labelling a safe arithmetic operation as dangerous, presumably as a result of its broader approach to overflow detection.

Race Conditions

By modelling all potential concurrent executions and demonstrating that no race conditions could arise, ZAN SCR demonstrated exceptional proficiency in validating contracts against race circumstances. Because of its comprehensive technique, the tool was able to identify every possible problem, however in “Crowdsale Contract with Race Condition” it mistakenly reported a synchronised operation for a race condition, leading to a false positive

Although VeriSol could identify more straightforward race problems, it failed to identify a flaw in “Voting Contract with Race Condition” which had complex time-based implications, leading to a false negative. This implies that VeriSol might find it difficult to investigate every concurrent scenario in detail, especially in contracts that are more complicated.

Access Control Issues

Through the effective verification of the contracts’ access control mechanisms, ZAN SCR made sure that access restrictions and permissions were applied correctly and in accordance with their specifications. Its thorough investigation did, however, result in a false positive in “Bank Contract with Access Control Vulnerability”, marking an appropriate access control mechanism as susceptible.

VeriSol detected simple problems with access control, but it overlooked a more complex vulnerability in “Bank Contract with Access Control Vulnerability” which had layered access permissions, which resulted in a false negative. This suggests that complex access control systems in contracts may not always be handled well by standard formal verification.

Denial of Service (DoS)

When it came to identifying and mathematically confirming the lack of DoS vulnerabilities, ZAN SCR excelled, especially when it came to execution restrictions and gas usage. Through the process of simulating different execution situations and analysing gas usage trends, ZAN SCR made sure that all contracts adhered to their specifications and no false positives or negatives recorded.

VeriSol was able to detect fundamental DoS vulnerabilities linked to gas, but it failed to detect “DoS with Unexpected Revert”, a more involved situation which included a conditional loop with variable gas consumption, which resulted in a false negative. Furthermore, VeriSol revealed its limitations in thoroughly assessing gas-related vulnerabilities when it labelled a safe, gas-efficient operation in “DoS with Unexpected Revert” as a possible DoS risk, resulting in a false positive.

Unchecked Call Return Values

ZAN SCR performed a great job of ensuring that every external call made inside the smart contracts was successfully verified, avoiding any potential vulnerabilities caused by call return values that were not evaluated. ZAN SCR could mathematically demonstrate that there were no unchecked operations by formally defining the expected behaviour of these calls. This allowed for perfect detection with no false positives or negatives

When VeriSol missed an unchecked call return in “UCRV in Contract Interaction” which involved several external calls in a single transaction, it resulted in a false negative. Additionally, a false positive was noted for the same contract when VeriSol inadvertently marked an appropriately handled external call for a contract as unchecked, demonstrating its shortcomings when managing complex contract interactions.

Timestamp Dependence

ZAN SCR was able to successfully confirm that there were no vulnerabilities related to timestamp reliance thanks to its AI-driven capabilities. ZAN SCR effectively made sure that neither of the two smart contracts relied on block timestamps for crucial operations, therefore generating no false positives or negatives.

When VeriSol was unable to detect a potential attack vector in the ”Lottery Time-Limited Auction” due to a non-linear usage of the timestamp, it was deemed a false negative. Furthermore, VeriSol’s inability to accurately assess time-based logic was demonstrated when it mistakenly identified a lawful timestamp use in ”Time-Limited Auction” as a vulnerability, resulting in a false positive.

Block Gas Limit Dependence

ZAN SCR performed flawlessly when confirming that smart contracts were immune to problems with block gas limits. ZAN SCR made sure that the contracts would not fail because of excessive gas use by mathematically analysing gas usage patterns and execution paths. This led to accurate detection with no false positives or negatives.

In the ”Gas-Intensive Loop Contract,” VeriSol produced a false negative by failing to identify a vulnerability where gas consumption varied depending on precise conditions. Furthermore, VeriSol generated a false positive in the same contract by misclassifying a gas-efficient operation as possibly susceptible to block gas limits.

Floating Point and Precision Issues

ZAN SCR successfully identified and confirmed that both smart contracts were free of floating-point and precision problems. Within the ”Incorrect Token Distribution Contract,” ZAN SCR effectively made sure that all computations followed the required specification, checking any possible vulnerabilities and ensuring that there were no false positives or negatives registered

Regarding the "Incorrect Token Distribution Contract," VeriSol overlooked a minor precision problem that could lead to an improper token allocation and thus resulted in a false negative. VeriSol's formal verification approach was arguably too simple to completely account for the complexities of floating-point arithmetic, especially in contracts where these operations were crucial.

Phishing and Spoofing

ZAN SCR was able to pinpoint areas where the contract can mistakenly trust external inputs or addresses, opening the door to possible spoofing attacks, by modelling the contract's external calls and verifying the integrity of those calls. Both vulnerabilities were discovered and there were no false positives or negatives because of the tool's capacity to thoroughly confirm that the contracts complied to their intended specifications.

In the "Fake Token Sale Contract," VeriSol overlooked a phishing vulnerability resulting in a false negative as it allowed a malicious contract to pose as a legitimate one. This problem most likely developed as a result of VeriSol's formal verification method not thoroughly examining all potential external contact scenarios, especially those requiring communication across contracts, which is more prone to involve spoofing.

9.3 Comparative Analysis

Thanks to its AI-powered formal verification capabilities, ZAN SCR truly outperformed VeriSol in finding and analysing vulnerabilities across a verity of smart contracts. Because of its comprehensive approach, ZAN SCR achieve a true positive rate of 100% and was able to detect subtle vulnerabilities like floating point and precision issues, as well as complex phishing and spoofing attacks, with increased accuracy in complicated scenarios. This careful consideration ensured no false negatives were recorded and a low false positive rate of 14.3%, which is especially critical in contracts where precision and external interacts are vital. For instance, VeriSol had difficulties with the "Incorrect Token Distribution Contract" and the "Fake Token Sale Contract," whereas ZAN SCR didn't and correctly highlighted their problems correctly.

Even though VeriSol was good at finding simple vulnerabilities, it had trouble with more complex situations. The complex arithmetic processes and external interactions seen in smart contracts make traditional formal verification approaches less suitable and resulted in the tool producing a true positive rate of 61.9%. This shortcoming was demonstrated by the increased rate of false negatives it produced, especially in contracts such as the "Fractional Price Calculation Contract," where it was unable to identify important weaknesses. More false positives were also produced by VeriSol with the tool achieving a false negative rate of 38.1%, which was highlighted in "Incorrect Token Distribution Contract," where normal activities were mistakenly interpreted as possible threats.

9.4 Usability and Processing Time

ZAN SCR demonstrated exceptional usefulness with its intuitive UI and efficient verification procedure, rendering it accessible to developers lacking extensive knowledge in formal verification. The tool's concise insights and suggestions enable swift vulnerability resolution. Because of its ability to automate difficult verification processes, it eliminated the need for a great deal of manual interpretation. This allowed it to integrate smoothly into development workflows and made it a useful option for practical applications. However, VeriSol presented a problem for users who were not familiar with these concepts because it required a deeper comprehension of formal verification. Particularly for complex contracts, the tool's output frequently required careful interpretation, which slowed down the verification process.

The average running time was 9.6 seconds per analysis for ZAN SCR, whereas VeriSol took an average of 1.8 minutes. The fast analysis of ZAN SCR attests to how well its AI-driven approach accomplishes formal verification tasks in an efficient manner. In relation to this, VeriSol uses more time in processing, which represents how most traditional methods are bound by computational overheads that make them not suitable for scenarios that demand quick, iterative security checks.

CHAPTER 10

Conclusion

The aim of this research was to determine how effectively AI-powered tools can perform compared to traditional tools in the detection of vulnerabilities in smart contracts written in Solidity and deployed on Ethereum-based blockchain platforms. In the process of the research, by investigating tools that used artificial intelligence such as ZAN SCR, Fuzzland and OlympiX, and traditional ones such as VeriSol, Echidna and Oyente, the paper detected considerable differences in terms of performance, precision, and efficiency. The findings reflect aspects related to the current state of security in smart contracts, with AI playing a significant role in increasing robustness in blockchain applications.

Among the many findings in this study, the superior accuracy of AI-powered tools is foremost. ZAN SCR and OlympiX showed true positive rates of 100% and 90.5%, respectively, compared to much lower rates found in traditional tools. For instance, VeriSol and Echidna, which are traditional tools, came with true positive rates of 61.9% and 52.4%, respectively. This disparity underlines the effectiveness of AI in the identification of vulnerabilities across a wide attack vector spectrum. AI-powered tools were good at identifying complex issues, such as phishing, spoofing, and advanced re-entrancy attack vectors that traditional tools usually failed to catch. These findings hint that machine learning algorithms and adaptive analysis techniques enable AI-powered tools to find not only the already known vulnerabilities but also self-improve towards the identification of new, previously unseen threats, thereby increasing overall security.

Another important observation from testing is that the false negative rate significantly decreases with AI-powered tools. For example, a few tools like ZAN SCR and Fuzzland remain very low or at zero false negative rates, which denotes their identification's reliability. Contrarily, 38.1% by VeriSol and 47.6% by Echidna denote the much higher rate of the traditional tools. Serious security risks are related to high false negative rates because they reflect missed vulnerabilities that might be further exploited in smart contracts. It is most likely that the low false negative rates of AI-powered tools reflect their ability to perform deeper analysis and spot more subtle patterns, enabling them to identify the broad range of security threats with much more precision.

Yet, in the wake of such an obvious difference, one should not forget that although these superior AI-powered tools provide better true positive and false negative rates, their false positive rates are just as comparable to traditional ones. Accordingly, both AI-powered and traditional machines yielded false positive rates at about 14-15%, which may consequently be translated to that AI does not significantly reduce the number of false alarms yet. This is an indication of the area where improvement could be likely to take place. Since this false-positive rate may result in resource and time consumption by the developers for investigating and fixing vulnerabilities that aren't actually present, improving the specificity of the AI models to keep the number of false positives as low as possible without compromising sensitivity remains critical for future improvements.

A major advantage that AI-powered tools showed in the formal verification category was with regard to the processing time. For example, the average processing time of ZAN SCR was 9.6 seconds, while that of VeriSol was 1.8 minutes. This makes the tool more appropriate in the context of real-time security checks and continuous integration pipelines where fast feedback is crucial. Shorter processing times will thus enable developers to find and fix vulnerabilities that much sooner, therefore closing the window of opportunity which may be capitalized on by an attacker.

The study has revealed the possibility of transforming the AI industry into one that will further increase the security of contracts on blockchain networks, such as Ethereum. AI-based tools are highly useful in a modern-day environment while developing blockchains, improving detection accuracy and reducing the number of false negatives through time-to-analyse quicker times. Yet, the challenge remains in reducing false positives, which points to space for innovation and improvement. Since blockchain technology is constantly in development, and smart contracts are becoming increasingly complex, further advancement and use of AI-powered security tools will continue to be integral to ensuring security and reliability for decentralized applications. This is where further research should be done to enhance the AI algorithms, with the goal of minimizing the number of false positives. Perhaps a hybrid approach could prove promising in which the strength of both the AI and the traditional approaches can offer their advantages, while also ensuring these can keep pace with the continuously changing landscape of blockchain security threats.

References

[1] A. Rosic. (2016). What is Blockchain Technology? A Step-by-Step Guide For Beginners. Accessed: June. 24, 2024. [Online]. Available: https://blockgeeks.com/guides/what-is-blockchaintechnology/
[2] Gurtu, A. and Johny, J., 2019. Potential of blockchain technology in supply chain management: a literature review. International Journal of Physical Distribution Logistics Management, 49(9), pp.881-900.
[3] Du, M., Chen, Q., Xiao, J., Yang, H. and Ma, X., 2020. Supply chain finance innovation using blockchain. IEEE transactions on engineering management, 67(4), pp.1045-1058.
[4] Khan, S.N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E. and Bani-Hani, A., 2021. Blockchain smart contracts: Applications, challenges, and future trends. Peer-to-peer Networking and Applications, 14, pp.2901-2925.
[5] Singh, A., Parizi, R.M., Zhang, Q., Choo, K.K.R. and Dehghantanha, A., 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. Computers Security, 88, p.101654.
[6] Cong, L.W. and He, Z., 2019. Blockchain disruption and smart contracts. The Review of Financial Studies, 32(5), pp.1754-1797.
[7] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," in Proc. Int. Conf. Princ. Secur. Trust, 2017, pp. 164–186.
[8] Wohrer, M. and Zdun, U., 2018, March. Smart contracts: security patterns in the ethereum ecosystem and solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 2-8). IEEE.
[9] Perez, D. and Livshits, B., 2019. Smart contract vulnerabilities: Does anyone care. arXiv preprint arXiv:1902.06710, pp.1-15.
[10] Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M. and Lee, H.N., 2022. Ethereum smart contract analysis tools: A systematic review. Ieee Access, 10, pp.57037-57062.
[11] Zhang, L., Wang, J., Wang, W., Jin, Z., Zhao, C., Cai, Z. and Chen, H., 2022. A novel smart contract vulnerability detection method based on information graph and ensemble learning. Sensors, 22(9), p.3581.
[12] Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X. and He, Q., 2021, January. Smart contract vulnerability detection using graph neural networks. In Proceedings of the Twenty-Ninth International Conference on International Joint

Conferences on Artificial Intelligence (pp. 3283-3290).
[13] He, D., Wu, R., Li, X., Chan, S. and Guizani, M., 2023. Detection of vulnerabilities of blockchain smart contracts. IEEE Internet of Things Journal, 10(14), pp.12178-12185.
[14] El Naqa, I. and Murphy, M.J., 2015. What is machine learning? (pp. 3-11). Springer International Publishing.
[15] Gupta, R., Tanwar, S., Al-Turjman, F., Italiya, P., Nauman, A. and Kim, S.W., 2020. Smart contract privacy protection using AI in cyber-physical systems: tools, techniques and challenges. IEEE access, 8, pp.24746-24772.
[16] Yu, X., Zhao, H., Hou, B., Ying, Z. and Wu, B., 2021, July. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In 2021 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.
[17] Jie, W., Chen, Q., Wang, J., Koe, A.S.V., Li, J., Huang, P., Wu, Y. and Wang, Y., 2023. A novel extended multimodal AI framework towards vulnerability detection in smart contracts. Information Sciences, 636, p.118907.
[18] He, D., Wu, R., Li, X., Chan, S. and Guizani, M., 2023. Detection of vulnerabilities of blockchain smart contracts. IEEE Internet of Things Journal, 10(14), pp.12178-12185.
[19] Huzenko, H. and Galchynsky, L., Mitigating the Ronin Protocol Vulnerability in the Context of RBAC Policy.
[20] Jie, W., Chen, Q., Wang, J., Koe, A.S.V., Li, J., Huang, P., Wu, Y. and Wang, Y., 2023. A novel extended multimodal AI framework towards vulnerability detection in smart contracts. Information Sciences, 636, p.118907.
[21] Wüst, K. and Gervais, A., 2018, June. Do you need a blockchain?. In 2018 crypto valley conference on blockchain technology (CVCBT) (pp. 45-54). IEEE.
[22] Yli-Huumo, J., Ko, D., Choi, S., Park, S. and Smolander, K., 2016. Where is current research on blockchain technology?—a systematic review. PloS one, 11(10), p.e0163477.
[23] Iansiti, M. and Lakhani, K.R., 2017. The truth about blockchain. Harvard business review, 95(1), pp.118-127.
[24] Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R. and Wang, F.Y., 2018, June. An overview of smart contract: architecture, applications, and future trends. In 2018 IEEE Intelligent Vehicles Symposium (IV) (pp. 108-113). IEEE.
[25] Mohanta, B.K., Panda, S.S. and Jena, D., 2018, July. An overview of smart contract and use cases in blockchain technology. In 2018 9th international conference on computing, communication and networking technologies (ICCCNT) (pp. 1-4). IEEE.
[26] Mezquita, Y., Valdeolmillos, D., González-Briones, A., Prieto, J. and Corchado, J.M., 2019. Legal aspects and emerging risks in the use of smart contracts based on blockchain. In Knowledge Management in Organizations: 14th International Conference, KMO 2019, Zamora, Spain, July 15–18, 2019, Proceedings 14 (pp. 525-535). Springer International Publishing.
[27] Jansen, M., Hdhili, F., Gouiaa, R. and Qasem, Z., 2020. Do smart contract languages need to be turing complete?. In Blockchain and Applications: International Congress (pp. 19-26). Springer International Publishing.
[28] Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B. and Roscoe, B., 2018, May. Reguard: finding reentrancy bugs in smart contracts. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeed-

ings (pp. 65-68).
[29] Sayeed, S., Marco-Gisbert, H. and Caira, T., 2020. Smart contract: Attacks and protections. Ieee Access, 8, pp.24416-24427.
[30] Praitheeshan, P., Pan, L., Yu, J., Liu, J. and Doss, R., 2019. Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605.
[31] Groce, A., Feist, J., Grieco, G. and Colburn, M., 2020. What are the actual flaws in important smart contracts (and how can we find them)?. In Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24 (pp. 634-653). Springer International Publishing.
[32] Ma, C., Song, W. and Huang, J., 2023, November. TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 947-959).
[33] Chen, W., Guo, X., Chen, Z., Zheng, Z. and Lu, Y., 2020, July. Phishing Scam Detection on Ethereum: Towards Financial Security for Blockchain Ecosystem. In IJCAI (Vol. 7, pp. 4456-4462).
[34] Chen, L., Peng, J., Liu, Y., Li, J., Xie, F. and Zheng, Z., 2020. Phishing scams detection in ethereum transaction network. ACM Transactions on Internet Technology (TOIT), 21(1), pp.1-16.
[35] Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A. and Hierons, R., 2018, March. Smart contracts vulnerabilities: a call for blockchain software engineering?. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 19-25). IEEE.
[36] He, D., Wu, R., Li, X., Chan, S. and Guizani, M., 2023. Detection of vulnerabilities of blockchain smart contracts. IEEE Internet of Things Journal, 10(14), pp.12178-12185.
[37] Deng, X., Beillahi, S.M., Minwalla, C., Du, H., Veneris, A. and Long, F., 2024, April. Safeguarding DeFi Smart Contracts against Oracle Deviations. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-12).
[38] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, National University of Singapore, Singapore. 2016. Oyente, ver. 1.0. Available: https://github.com/ethereum/oyente
[39] Consensys. Mythril: Security analysis tool for EVM bytecode. 2024 GitHub. Available at: https://github.com/Consensys/mythril
[40] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, Trail of Bits, New York, NY, USA. 2018. Manticore, ver. 0.3.4. Available: https://github.com/trailofbits/manticore
[41] Crytic. Echidna: Ethereum smart contract fuzzer. 2024 GitHub. Available at: https://github.com/crytic/echidna [42] R. Shinde, M. Zhang, A. B. Windsor, and M. Ranganathan, Microsoft Research, Redmond, WA, USA. 2019. VeriSol, ver. 0.1.1. Available: https://github.com/microsoft/verisol
[43] Xiong, W. and Xiong, L., 2019. Smart contract based data trading mode using blockchain and machine learning. IEEE Access, 7, pp.102331-102344.
[44] Gupta, R., Patel, M.M., Shukla, A. and Tanwar, S., 2022. Deep learning-based malicious smart contract detection scheme for internet of things environment. Computers Electrical Engineering, 97, p.107583.
[45] Mat Rahim, S.R., Mohamad, Z.Z., Abu Bakar, J., Mohsin, F.H. and Md Isa, N., 2018. Artificial intelligence, smart contract and islamic finance. Asian Social Science, 14(2), p.145.

[46] Deebak, B.D. and Fadi, A.T., 2021. Privacy-preserving in smart contracts using blockchain and artificial intelligence for cyber risk measurements. Journal of Information Security and Applications, 58, p.102749.
[47] Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R. and Wang, F.Y., 2018, June. An overview of smart contract: architecture, applications, and future trends. In 2018 IEEE Intelligent Vehicles Symposium (IV) (pp. 108-113). IEEE.
[48] OlympiX, ver. 1.0. Available at: https://www.olympix.ai/
[49] ZAN SCR Formal Verification Engine, Available at: https://docs.zan.top/docs/formal-verification
[50] Fuzzland. Fuzzland: Smart Contract Fuzzing Tool. Available at: https://fuzzland.com