National College of Ireland

Optimizing Detection of Reentrancy attacks in Solidity Smart Contracts

MSc Industrial Internship
MSc In Cybersecurity

Mayank Sharma
Student ID: 21156913

School of Computing
National College of Ireland

Supervisor: Prof. Vikas Sahni

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Mayank Sharma		
Student ID:	21156913		
Programme:	Msc. In Cybersecurity	**Year:**	2022-2023
Module:	Msc Industrial Internship		
Supervisor:	Prof. Vikas Sahni		
Submission Due Date:	06-JAN-2023		
Project Title:	Optimizing Detection of Reentrancy attacks in Solidity Smart Contracts		
Word Count:	6125	**Page Count:**	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Mayank Sharma

Date: 04-JAN-2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Optimizing Detection of Re-entrancy attacks in Solidity Smart Contracts

Mayank Sharma
21156913

Abstract

Smart contracts, which are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code, have the potential to automate and improve efficiency in various industries such as supply chain management, financial transactions, and legal agreements by storing the contract on a distributed ledger. However, vulnerabilities in the code of smart contracts can be exploited by malicious actors, leading to unauthorized access to funds or sensitive information, disruption of contract execution, and other harmful effects. One type of attack on smart contracts is the reentrancy attack, in which a contract function is repeatedly called before its execution is complete, potentially allowing for the manipulation of data or draining of funds.

The proposed solution combines manual testing with static code analysis and the use of tools such as Hardhat and Slither to optimize and maximize the detection of reentrancy attacks and other vulnerabilities in smart contracts. By following this approach, developers can help to ensure the reliability and trustworthiness of their contracts and protect against attacks. This research aims to provide a solution for detecting reentrancy attacks in smart contracts that involve a combination of manual testing using the Hardhat test runner and static analysis with the Slither tool. Initial evaluation of this approach has shown promising results in efficiently detecting vulnerabilities and mitigating risks prior to deployment. Further research is needed to fully assess the effectiveness of this solution and explore potential improvements.

1 Introduction

One type of distributed ledger, a blockchain, is utilized to manage assets that are shared by users. These assets are transferred according to the rules that are encoded in a smart contract. The transfers take place within immutable transactions that are logged on the blockchain.

Ethereum is one of the most widely used blockchains that enables smart contracts. The market value of Ethereum was approximately $130 billion in January 20211. On Ethereum, smart contracts allow users and other contracts to interact with them by calling functions in the contracts. An execution cost known as gas is required to carry out the transaction on a smart contract. The cryptocurrency of Ethereum, ether (ETH), is used to pay for gas[1].

[1]https://www.csc.kth.se/~dilian/Papers/ease21.pdf

Ethereum is a decentralized, open-source blockchain platform that enables the creation of smart contracts. It allows developers to build and deploy decentralized applications, or dApps, on its platform.

The most widely used tool for creating Ethereum smart contracts today is Solidity, which was created by the Ethereum project's developers explained by (Kaleem, et al., 2020). Additionally for writing high-level smart contracts, Solidity has been utilized vastly by the Developers. However, smart contracts developed in Solidity have security flaws wherein different attacks on Ethereum-based projects have been widely reported exploited described by (Chen, et al., 2020).

The most well-known instance of a smart contract attack is the one on "The DAO (decentralized autonomous organizations)" in June 2016 described in research by (Morrison, et al., 2020). Reentrancy is a vulnerability that was exploited in the attack to steal more than 60 million US dollars worth of ether. Reentrancy occurs when the main contract calls an external contract, which then contacts the calling contract once more in a single transaction. Due to the transparency of the Ethereum blockchain, it is possible to continuously launch an attack using the deployed contracts' flaws.

1.1 Research Objective and Question

The goal of this research is to the detection reentrancy attacks in solidity smart contracts.

The research objective of this study was to investigate and identify potential vulnerabilities and security issues in smart contracts. The research question for this study was: What are the most effective methods for detecting and mitigating vulnerabilities and security issues in smart contracts?

1.2 Motivation

The motivation for researching the detection of reentrancy attacks in smart contracts stems from the fact that these attacks can have significant consequences for the security and integrity of blockchain systems. Reentrancy attacks involve a malicious contract calling an external contract multiple times before the external contract has completed its execution, leading to a state of infinite loop and potentially allowing the attacker to drain the victim contract of its resources (Buterin, 2014).

Reentrancy attacks can have serious implications for the stability and security of blockchain systems, as they can lead to the loss of funds or the disruption of critical services. For example, the infamous DAO attack of 2016, in which a hacker exploited a reentrancy vulnerability to drain the Ethereum-based Decentralized Autonomous Organization of millions of dollars, led to a hard fork in the Ethereum network and highlighted the need for robust security measures to prevent such attacks (Buterin, 2016). Another such attack lead by a group or individual carried out two "reentrancy attacks" on the Uniswap and Lendf.me cryptocurrency platforms,

stealing more than $25 million. The attacks were made possible by a known vulnerability in Uniswap's ERC777-token, which allows for repeated withdrawal of funds before the initial transaction is approved or declined. Uniswap lost between $300,000 and $1.1 million in imBTC tokens, while Lendf.Me lost more than $24 million.

Given the potential consequences of reentrancy attacks, it is important for developers to have effective tools and methods for detecting and mitigating these vulnerabilities in smart contracts. This motivation drives the research on the detection of reentrancy attacks and the development of solutions such as the proposed combination of manual testing and static analysis.

1.3 Paper Outline

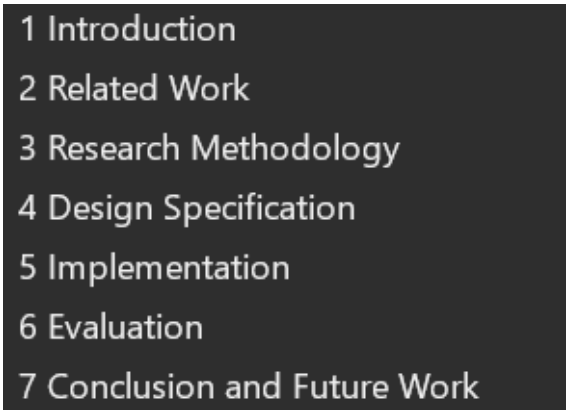

Figure 1: Paper structure

2 Related Work

A smart contract is a type of internet protocol that carries out scheduled tasks without the requirement for supervision by a reliable third party. *"Computer program stored in a distributed ledger system wherein the outcome of any execution of the program is recorded on the distributed ledger"* are called smart contracts as defined by ISO[2]. In addition to facilitating wallet functionality, voting, bidding, market forecasting, supply chain tracking, and other purposes, smart contracts broadened the applicability of blockchain.

The Ethereum network, which uses the Solidity programming language, is frequently threatened by reentrancy attacks. The attack occurred when an adversary uses an external call of a smart contract to force the contract to execute more code by calling back to itself using a fallback function before updating the final state of the contract addressed by (Alkhalifah, et al., 2021).

[2]https://www.iso.org/obp/ui/fr/

Nick Szabo developed the concept of smart contracts in 1996 has been illustrated by (Kim & Tai-hoon, 2016). It has been stated that smart contracts are lines of code that are implemented on a blockchain network and automatically carry out transactions in a transparent and decentralized method. It has been observed by (Huang, et al., 2017) a smart contract cannot be changed once it has been deployed, and there are very few ways to deal with errors that occur during contract execution.

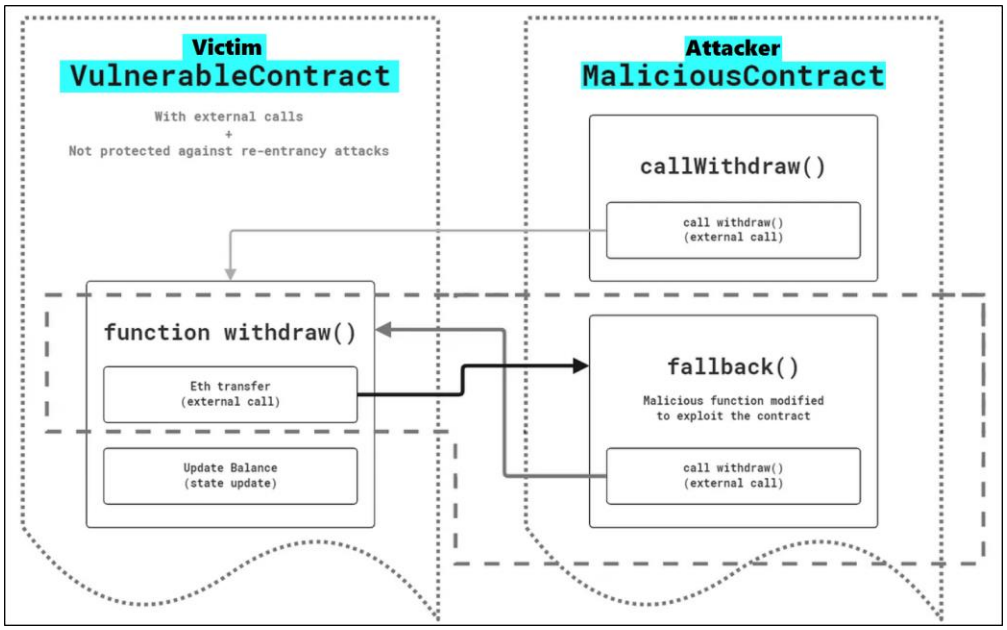

Figure 2: Reentrancy attack

2.1 Smart contracts

Smart contracts, a technology in the field of blockchain that encodes the terms of an agreement into code and stores it on a decentralized network, have been utilized to automate and streamline various processes, including supply chain management, financial transactions, and legal agreements. The automation and enforceability of contracts, made possible by smart contracts, eliminate the need for intermediaries and reduce the risk of errors or fraud. However, vulnerabilities in the code of smart contracts can be exploited by hackers and malicious actors, leading to unauthorized access to funds or sensitive information, disruption of contract execution, and other harmful effects. One type of attack on smart contracts is the reentrancy attack, in which a contract function is repeatedly called before its execution is complete, potentially allowing for the manipulation of data or draining of funds discussed by (Buterin, 2014). Other vulnerabilities that may be exploited include lack of input validation, race conditions, and denial of service attacks.

The consequences of a successful attack on a smart contract can include financial loss, reputation damage, and legal liability, and may be motivated by a desire to steal funds, disrupt operations, or gain access to sensitive information.

2.1.1 Smart contract vulnerabilities

Several types of vulnerabilities can occur in smart contracts, and the most impactful ones can have serious consequences for the contract and its stakeholders. Some common types of vulnerabilities in smart contracts include:

- Reentrancy attacks: These attacks allow an attacker to repeatedly call a contract function before it has completed execution, potentially allowing the attacker to drain funds or manipulate data explained by (Buterin, 2014).

- Race conditions: These vulnerabilities occur when a contract has multiple concurrent processes that are not properly synchronized, potentially allowing an attacker to exploit the race condition to manipulate data or execute unintended actions.

- Denial of service attacks: These attacks involve overwhelming a contract with a large number of requests or transactions, potentially causing it to malfunction or become unavailable

- Unchecked call/send: If a contract does not properly check the return value of a call or send operation, it may be vulnerable to attack through the injection of malicious code

Developers need to be aware of these and other vulnerabilities described by (Nguyen, et al., 2021) and (Mense & Flatscher, 2018) to design and implement effective security measures to prevent attacks. By thoroughly testing and analyzing smart contracts, developers can help to ensure their reliability and trustworthiness and minimize the risk of vulnerabilities being exploited.

2.2 Importance of Reentrancy attack

Reentrancy attacks are a type of vulnerability that allows an attacker to repeatedly call a function in a smart contract before the initial transaction has been completed. This can allow the attacker to drain funds from the contract or manipulate its state in ways that were not intended by the developer. Reentrancy attacks are a major concern for smart contracts because they can be difficult to detect and can have significant consequences for the contract and its users. In the Decentralized Application Security Project (DASP) Top 10 of 2018, reentrancy attacks were ranked as the second most common type of vulnerability in smart contracts, highlighting the importance of addressing this issue. To mitigate the risk of reentrancy attacks, developers need to follow best practices and security standards when coding and testing their smart contracts, such as the OWASP Secure Coding Practice and Solidity Smart contract security consideration. By carefully considering the potential vulnerabilities of their smart contracts and taking steps to address them, developers can help ensure the security and reliability of their applications.

2.3 Types of Reentrancy Attacks

When an external function invokes another insecure contract and gave an attacker control of that insecure contract and opens a vulnerability. A reentrancy attack would be carried out by the adversary by recursively calling the vulnerable function and consuming all ether until all the gas is consumed and the adversary would exhaust all ether in the victim's smart contract. "gas" refers to the extra cost associated with completing a smart contract or transaction on the blockchain network has been reported by (Atzei, et al., 2017).

2.3.1 Single Reentrancy Attacks

When the vulnerable function is identified by the attacker and the attacker keeps calling that function recursively is known as a single reentrancy attack as described in the research by (Samreen & Alalfi, 2020).

2.3.2 Cross-function Attacks

A cross-function reentrancy attack is achieved when an attacker exploited the state of a vulnerable function that shares a state with another function. Hence, it is hard to detect as described in the research (Samreen & Alalfi, 2020).

A hacker found a vulnerability in the DAO's code and was able to steal 3.6 million Ethereum into his account, which caused the price of ether to plunge and discouraged investors in the community reported by (Sayeed, et al., 2020). This is the famous reentrancy attack that was carried out by adversaries in 2016. This research talks about the automating detection of reentrancy attacks using slither modules and hardhat in solidity smart contracts.

2.4 Limitations

The various techniques used to guard against the reentrancy vulnerability in smart contracts are described here. This includes using secure coding practices, such as properly handling input validation, using secure libraries and frameworks, and implementing strict access controls. This research is focused on automating the static analysis for detecting reentrancy vulnerabilities in solidity smart contracts.

Additionally, the use of smart contract testing and analysis tools, such as Mythril and Oyente, can help identify potential vulnerabilities in contract code and provide recommendations for remediation.

2.4.1 Mythril

It offers a unique examination of smart contract vulnerabilities based on symbolic code execution which is used to find security vulnerabilities with smart contracts. The accuracy of Mythril and other analysis tools is limited by the quality and completeness of their vulnerability detection database and ruleset. Smart contracts created for EVM-compatible blockchains like Ethereum and Tron can be scanned by Mythril using EVM bytecode to find cybersecurity flaws as mentioned in the research by (Sayeed, et al., 2020).

2.4.2 Oyente

A similar and better tool that works on symbolic execution, one which can identify security flaws in smart contracts. The character of an execution path is symbolically represented by a

mathematical formula in the symbolic execution approach. It also determines whether the new formula and formulae with common bugs can both be used simultaneously, Oyente compares the two formulas as described by (Luu, et al., 2016)

2.4.3 SolidityCheck

It is another open-source tool based on regular expressions and program instrumentation. Smart contract files with their Solidity source code are sent to SolidityCheck. The source code is first prepared so that regular expressions may be accessed. Following that, the problem statement is located by matching the bug statements with particular regular expressions. SolidityCheck combines software instrumentation to guard against integer overflow flaws and reentrancy flaws as described by (Zhang, et al., 2019) The fact that it only uses regular expressions means that flaws like cross-reentrancy do not get recognized. These patterns are based on previous attacks and known vulnerabilities, making them limited and difficult to generalize.

RESULTS ON THE 50 VULNERABLE CONTRACTS IN DATASET2

	Simple Reentrancy	Modified Reentrancy	Buggy-locked Reentrancy	Total
Oyente	20/20	10/10	15/20	45/50
Mythril	17/20	4/10	8/20	29/50
Slither	20/20	5/10	15/20	40/50
ReDefender	20/20	10/10	20/20	50/50

Figure 3: Tool comparison results

RESULTS ON THE 40 CONTRACTS IN DATASET3

	Send	Transfer	Secure Mutex Lock	Secure Time Lock	Infeasible Path	Total
Oyente	0/5	0/5	0/10	10/10	10/10	20/40
Mythril	0/5	0/5	2/10	4/10	8/10	14/40
Slither	5/5	5/5	5/10	10/10	10/10	35/40
ReDefender	0/5	0/5	0/10	0/10	0/10	0/40

Figure 4: Tool comparison results

Based on the existing research Slither has been selected as the static analysis tool in this research due to its demonstrated accuracy and efficiency in detecting vulnerabilities in smart contracts. In comparison to other tools such as oyente, Mythril, and solidity-check, Slither has consistently demonstrated superior performance in detecting a wide range of vulnerabilities, including reentrancy attacks. In addition to its high accuracy, Slither also utilizes the SlithIR technology, which allows for efficient analysis of smart contracts and reduces the number of false positives that may be produced by other tools described by (Feist, et al., 2019). Hardhat, a test runner tool, was also utilized in this research to run unit tests against the solidity smart contract and effectively detect reentrancy attacks through the creation of specific test cases. To address these vulnerabilities, the combination of static code analysis, manual testing, and the use of tools such as Hardhat and Slither can be utilized by developers and security researchers to detect and prevent attacks said by (Sklaroff, 2017) By taking a proactive approach to security, the reliability and trustworthiness of smart contracts can be ensured and their potential benefits maximized.

2.5 Research Niche

Table 1: Research Niche

Related Work	Strength	Limitation	No. of Citations
(Alkhalifah, et al., 2021)	Comprehensive combines static and dynamic analysis	Relies on analysis tools, does not detect all vulnerabilities	9
(Atzei, et al., 2017)	Provides a comprehensive overview of attacks on Ethereum smart contracts	Does not cover all types of attacks.	1457
(Buterin, 2014)	Provides a flexible and powerful platform for building decentralized applications	scalability issues, vulnerable to certain types of attacks.	3967
(Chen, et al., 2020)	Covers three aspects of Ethereum security	Not up to date.	274
(Chen, et al., 2020)	Generic, efficient, easily extendable, compatible with multiple blockchains	Requires deep understanding of blockchain internals to develop new detection apps	55
(Feist, et al., 2019)	Checks for a wide range of vulnerabilities, easy to use.	Produce false positives or false negatives.	211
(Huang, et al., 2017)	Decentralized, provides trust in IoT data exchange, transparent, auditable	Requires in-depth knowledge of applications in IoT	102
(Kaleem, et al., 2020)	Comparison of security vulnerabilities between Solidity and Vyper	Not updated with new attack vectors	24
(Kim & Tai-hoon, 2016)	Convenient, fast, no intermediaries or fees, easily transferable across borders	Subject to digital security risks and not widely accepted	29
(Koscher, et al., 2010)	Demonstrates the fragility of modern automotive systems, highlights potential risks	Not up to date.	2172
(Luu, et al., 2016)	Identifies new security problems	Not up to date.	1806
(Mense & Flatscher, 2018)	Compares code analysis tools	Not up to date.	79
(Morrison, et al., 2020)	Challenges of governance in blockchain-based organizations	Not be applicable to all blockchain-based organizations	40

(Nguyen, et al., 2021)	Can fix common vulnerabilities in smart contracts, proven to be sound	Introduce minor overhead with new attack vectors	21
(Rodler, et al., 2018)	Protects existing contracts, accurate, minimal overhead	Only protects against reentrancy attacks.	158
(Salehi, et al., 2022)	Upgradeability patterns in smart contracts and a measurement framework to evaluate their use	Limited to the Ethereum blockchain, may not cover all upgradeability patterns	0
(Samreen & Alalfi, 2020)	Combines static and dynamic analysis, high performance, low false positives	Lacks Manual testing	32
(Sayeed, et al., 2020)	Comprehensive overview of smart contract vulnerabilities and attacks	Not up to date	140
(Sharma, et al., 2022)	Explores security perceptions and practices of smart contract developers	A qualitative, exploratory study with a small sample size	3
(Sklaroff, 2017)	Technologically advanced integrates with modern systems, and offers a range of operations.	Require human intermediaries	331
(Zhang, et al., 2019)	Efficient, effective, classifies problems in Solidity	Limited to Solidity, may not detect all problems	19

3 Research Methodology

During this research, the selection and use of relevant keywords were carefully considered in order to effectively investigate the vulnerabilities and potential attack vectors of smart contracts. These keywords included: "smart contracts," "Ethereum," "blockchain," "vulnerabilities," "security," "reentrancy attacks," "static code analysis," "manual testing," "Hardhat," and "Slither." By including these keywords in the research, a comprehensive understanding of the topic was gained.

By using these and other relevant keywords, it is possible to locate a variety of research papers and other resources that can provide valuable insights into the vulnerabilities and potential attack vectors of smart contracts. By carefully reviewing the proposed solution in this research, a developer can gain a better understanding of the various techniques and tools available for detecting reentrancy attacks and other vulnerabilities in smart contracts. This can help the developer design and implement effective testing and security measures for their smart

contracts before going on the Ethereum network. It is generally not possible to directly modify or delete a smart contract once it has been deployed to a blockchain due to the permanent, immutable nature of the blockchain addressed by (Buterin, 2014). However, vulnerabilities in a deployed smart contract can be addressed by deploying a new contract that includes the necessary fixes, a process known as contract upgrading or contract migration mentioned by (Salehi, et al., 2022). While contract upgrading can be a viable solution, it is important to carefully consider the costs and challenges involved, including fees for deploying the new contract, coordination and communication with users, and potential confusion or mistrust among users discussed by (Sklaroff, 2017).

3.1 Combined Approach mitigating reentrancy attack in solidity smart contracts

3.1.1 Manual Unit Testing

Hardhat is a set of tools for Ethereum development that includes a test runner and a task runner. The test runner can be used to detect reentrancy attacks by running tests on a smart contract to ensure that it is resistant to such attacks discussed by (Sharma, et al., 2022). Chai and Mocha testing frameworks provide a range of functions and assertions for verifying the behavior of a contract and detecting potential vulnerabilities.

To detect a reentrancy attack, the test runner in Hardhat verifies that the smart contract correctly handles reentrancy by checking for the presence of a guard condition. A guard condition is a check that prevents the contract from being called again before the previous call has been completed and is an important defense against reentrancy attacks mentioned by (Koscher, et al., 2010). If a guard condition is not present, the test runner may flag the smart contract as vulnerable to reentrancy attacks.

In addition to checking for the presence of a guard condition, the Hardhat test runner also uses other techniques to detect reentrancy attacks, such as tracking the state of the contract or monitoring the balance of external contract calls mentioned in research by (Sharma, et al., 2022). By utilizing these and other techniques, the Hardhat test runner helps to ensure that a smart contract is resistant to reentrancy attacks and secure against this type of attack.

3.1.2 Static Code Testing

Slither is a static code analyzer for Ethereum smart contracts that can be used to detect vulnerabilities and security issues in the code. It achieves this by performing a thorough analysis of the code using techniques such as taint analysis and control flow analysis, which help to identify potential vulnerabilities and security flaws (Feist, et al., 2019).

Taint analysis is a technique that tracks the flow of data through a program and identifies potentially sensitive or vulnerable data By tracking the flow of data, Slither can detect instances where sensitive data may be at risk of being accessed or modified by an attacker.

Control flow analysis, on the other hand, is a technique that examines the flow of control through a program and identifies potential vulnerabilities and security issues. By analyzing the control flow of the program, Slither can identify instances where the program may be vulnerable to attacks such as race conditions or denial of service attacks discussed in research by (Feist, et al., 2019).

The proposed solution in this research utilizes a combination of manual testing with Hardhat and static analysis with Slither to effectively detect and mitigate vulnerabilities in smart contracts. By utilizing Hardhat to run unit tests against the solidity smart contract, developers can efficiently identify and fix vulnerabilities through the creation of specific test cases. This is followed by the use of Slither, a static analysis tool that utilizes SlithIR technology to efficiently analyze smart contracts and reduce the number of false positives. By combining these two approaches, developers can effectively detect and fix vulnerabilities in smart contracts before deployment on the blockchain network. It is worth noting that Hardhat operates on its network, allowing for the compilation and testing of contracts without the risk of affecting the live blockchain network. This makes it a useful tool for detecting vulnerabilities early in the development process.

4 Design Specification

```
define static_reentrancy_scan():
    run slither with the appropriate arguments
    if the slither command returns a non-zero exit code:
        print "Reentrancy vulnerabilities detected"
    else:
        print "No reentrancy vulnerabilities detected"

define manual_reentrancy_scan():
    run npx hardhat test
    if the command returns a non-zero exit code:
        print "Reentrancy vulnerabilities detected"
    else:
        print "No reentrancy vulnerabilities detected"

prompt the user to choose which scan to run
if the user chooses the manual scan:
    run the manual_reentrancy_scan() function
else if the user chooses the static scan:
    run the static_reentrancy_scan() function
else:
    print "Invalid scan type"
```

Figure 5: Design pseudocode

In this research, a python script has been developed that utilizes the Hardhat test runner and the Slither static code analyzer to detect reentrancy attacks and identify vulnerabilities and security issues in a smart contract, respectively. The Hardhat test runner is used to verify the resistance of the smart contract to reentrancy attacks by checking for the presence of a guard condition. Slither, on the other hand, employs various techniques such as taint analysis and control flow analysis to thoroughly analyze the code and identify potential vulnerabilities and security flaws. The use of these tools is based on the creation of test cases specifically designed for the given smart contract.

It has been recommended to combine static analysis with manual testing to increase the likelihood of detecting all potential vulnerabilities and security issues in smart contracts described by (Koscher, et al., 2010). Performing manual testing prior to static testing involves the creation of test cases by a developer who has thoroughly analyzed the parameters and functions in the contract and is designed to uncover potential vulnerabilities and security issues that may not be detected through static analysis alone The proposed solution also utilizes the Hardhat test runner, which is a set of tools for Ethereum development that includes a test runner, to detect reentrancy attacks The test runner can verify the resistance of a smart contract to reentrancy attacks by checking for the presence of a guard condition, which is a check that prevents the contract from being called again before the previous call has completed. By following this approach, the proposed solution aims to optimize and maximize the detection of all potential reentrancy vulnerabilities in the solidity smart contract.

5 Implementation

Smart contracts have become increasingly popular and have garnered widespread attention as a revolutionary technology in various industries. However, the immutability and transparency of smart contracts also make them vulnerable to cyber attacks. Reentrancy attacks, in particular, have been a major concern for developers and researchers, as they allow an attacker to repeatedly call a contract's function and drain its funds highlighted by (Chen, et al., 2020).

To address this issue, the proposed solution in this research combines the use of manual testing with Hardhat and static code analysis with Slither to effectively detect and mitigate vulnerabilities in smart contracts. Hardhat is a tool for Ethereum development that includes a test runner, allowing for the creation of specific test cases to identify vulnerabilities and security issues. Slither, on the other hand, utilizes SlithIR technology to efficiently analyze smart contracts and reduce the number of false positives as described by (Feist, et al., 2019). By combining these two approaches through the implementation of a Python script, the proposed solution aims to automate the testing process and provide a comprehensive and efficient method for detecting and mitigating vulnerabilities in smart contracts before deployment on the blockchain network.

The output of the above script will depend on the results of the reentrancy scans. If the manual reentrancy scan or the static reentrancy scan detects a reentrancy vulnerability in the contract,

the script will print "Reentrancy vulnerabilities detected". If no vulnerabilities are detected, the script will print "No reentrancy vulnerabilities detected".

The proposed solution for detecting and mitigating reentrancy attacks in Solidity smart contracts was implemented through a combination of static analysis using the Slither code analyzer and manual testing using the Hardhat test runner. Slither code analyzer was used to perform a static analysis of the target smart contract. This involved running the slither command followed by the path to the contract and the '--detect' all kinds of reentrancy attacks, which instructed Slither to specifically search for reentrancy vulnerabilities. If Slither detected any vulnerabilities, they were reported in the output but it also results in a few false positives as mentioned in related work. Therefore, it is important to carefully review the results of static analysis and additional run another test for the validity of any reported vulnerabilities.

Hence, to cover this, the proposed solution includes manual testing using the Hardhat test runner. The Hardhat test runner was used to detect reentrancy attacks in a Solidity smart contract by creating a test case in a JavaScript file and running it using the hardhat test command followed by the path to the contract. The proposed Javascript unit test was created and run against the vulnerable contracts, which also does an additional check along with the existing logic mentioned in this research that detects the reentrancy vulnerability in the smart contract. The test runner reported success and printed a message indicating that the contract passed the test. By using the Hardhat test runner in this way, the reentrancy vulnerabilities in the smart contract were successfully detected and appropriate measures could be taken to mitigate them. This helped to ensure the security and reliability of the contract on the blockchain, mitigating the potential for costly and devastating consequences due to vulnerabilities and security issues.

Overall, this script can be a useful tool for developers to detect and mitigate reentrancy attacks in their smart contracts, thereby improving the security and reliability of their contracts on the blockchain.

6 Evaluation

Four vulnerable contracts were created to evaluate the solution. Out of these four contracts, one contract was used to create a custom test case that would run via hardhat and detect the reentrancy attack based on the logic defined in the test case. The other three vulnerable contracts were run against a static analyzer, which successfully detected all the reentrancy attacks without fail.

The test results demonstrate that detecting vulnerabilities before deployment can be optimized by adding manual testing to the process as described by (Sklaroff, 2017). This approach and solution can be used by blockchain developers to test their smart contracts for reentrancy attacks. According to the state-of-the-art discussed in related work, static testing approaches come with false positives, and adding a manual testing approach can significantly reduce false positives.

Table 2: Evaluation Summary

Vulnerable smart contract	Reentrancy detected	Unit Test Created
GasWallet.sol	Yes	Yes
vulnContract1.sol	Yes	No
vulnContract2.sol	Yes	No
vulnContract3.sol	Yes	No

The GasWallet contract was manually tested using a custom test case running in hardhat specifically designed to detect reentrancy attacks. The test was successful in detecting the vulnerability present in the contract. Additionally, three other contracts (vulnContract1, vulnContract2, and vulnContract3) were evaluated using static analysis with the help of the Slither tool and its built-in detectors. These evaluations were also successful in detecting the reentrancy vulnerabilities present in the contracts.

6.1 Test 1

The process involves running the hardhat test followed by the slither static code analysis, which enables the detection of reentrancy attacks in vulnerable smart contracts.

A custom test case was created to test for the presence of reentrancy vulnerabilities in GasWallet.sol using the APT.sol attacker contract. The APT.sol contract, which serves as an attacker, was also created and used in the test case to successfully test the reentrancy vulnerability. The test case involved attempting to withdraw funds from the victim contract over the hardhat network. The results of the test were used to determine whether the GasWallet.sol contract was vulnerable to reentrancy attacks.

```
Enter 'manual/m' to run the manual reentrancy scan or 'static/s' to run the static reentrancy scan or both/b: b


  Compile and Deploy Smart contracts
    Check for deposit/withdraw functions
      ✓ Withdraw functions exist (41ms)
      ✓ Deposit function exists
Attack Started----!
GAS Balance: 190.0
Attacker's GAS Amount: 9999.9977697988691665
Amount transferred to APT---!
GAS Balance: 0.0
Attackers's GAS Amount: 10189.997347135693789145

      ✓ Reentracy attack detected (145ms)


  3 passing (1s)


Reentrancy in Swissbank.withdrawmoney() (../../smartcontracts/Reentrancy-attack-Smart-Contract-Security-master/Reentr
ancy-attack-Smart-Contract-Security-master/contracts/vulnContract3.sol#11-19):
        External calls:
        - (sent) = msg.sender.call{value: bal}() (../../smartcontracts/Reentrancy-attack-Smart-Contract-Security-mast
er/Reentrancy-attack-Smart-Contract-Security-master/contracts/vulnContract3.sol#15)
        State variables written after the call(s):
        - balances[msg.sender] = 0 (../../smartcontracts/Reentrancy-attack-Smart-Contract-Security-master/Reentrancy-
attack-Smart-Contract-Security-master/contracts/vulnContract3.sol#18)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
```

Figure 6: Reentrancy Vulnerability detected by Manual and Static testing

6.2 Test 2

Static analysis tool used to detect vulnerabilities of reentrancy attacks in vulnerable smart contracts. The results of the analysis provided the specific line numbers where these vulnerabilities were found in the contracts.

vulnContract3.sol is detected with reentrancy vulnerability successfully.

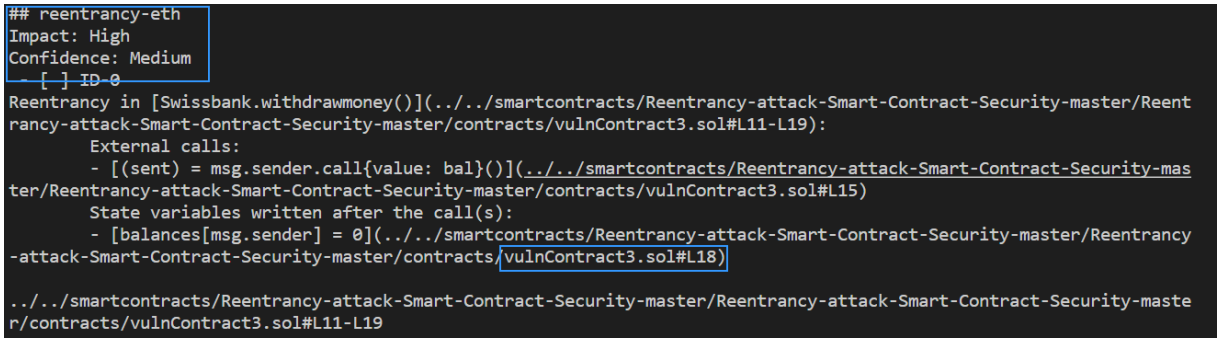

Figure 7: Reentrancy Vulnerability Detected

6.3 Test 3

vulnContract2.sol is detected with reentrancy vulnerability successfully.

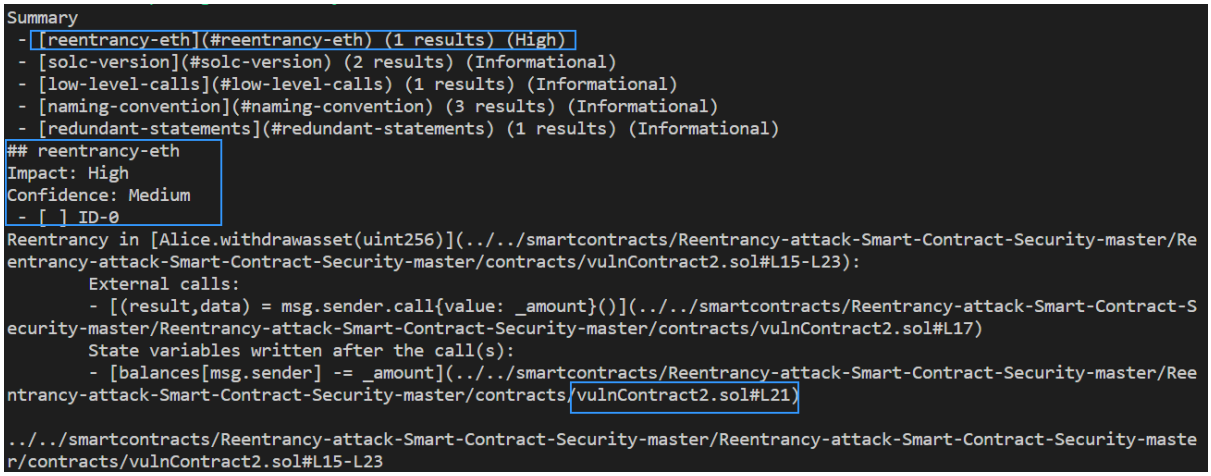

Figure 8: Reentrancy Vulnerability Detected

6.4 Test 4

vulnContract1.sol is detected with reentrancy vulnerability successfully.

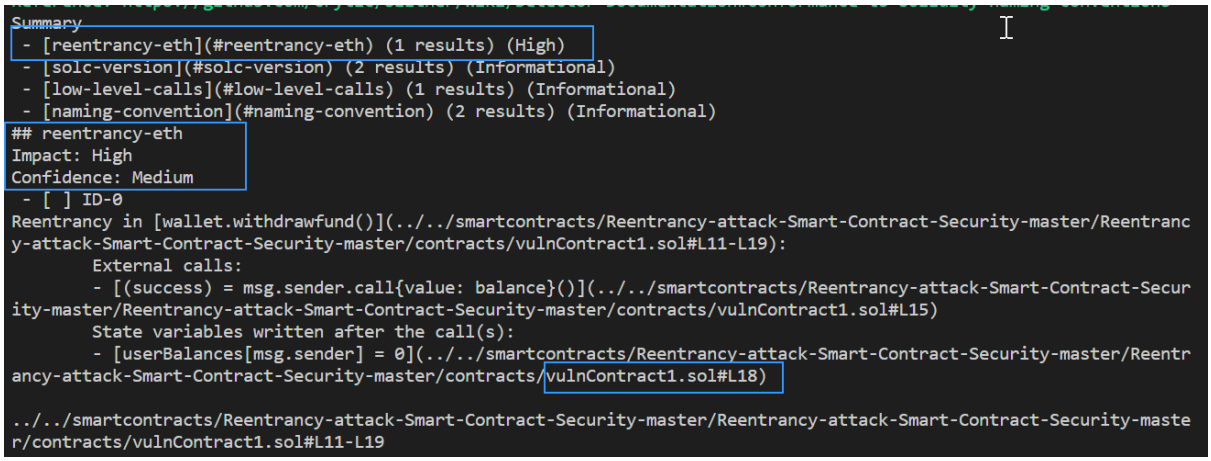

Figure 9: Reentrancy Vulnerability Detected

6.5 Discussion

The proposed solution for detecting reentrancy attacks in smart contracts involves a combination of manual testing using the Hardhat test runner and static code analysis using the Slither tool. By performing both manual testing and static analysis, the proposed solution aims to increase the likelihood of detecting all potential reentrancy vulnerabilities and security issues in smart contracts.

One potential limitation of the proposed solution is that manual testing can be time-consuming and may not be feasible for large or complex contracts. Additionally, manual testing may not be able to uncover all vulnerabilities, as it relies on the developer's ability to identify and build test cases for potential attack vectors.

In order to enhance the security of smart contracts with complex architecture, it is recommended to incorporate the enterprise version of Mythx. This version maintains a smart contract security repository which is mapped to common weakness enumeration and provides guidelines for the secure development of smart contracts using Solidity. One of the advantages of Mythx is its ability to identify vulnerabilities that may not be detectable through static analysis or manual testing. In addition, dynamic analysis techniques such as fuzz testing can be utilized to expose potential vulnerabilities. These improvements can help to optimize and maximize the detection of all potential vulnerabilities and security issues in smart contracts.

7 Conclusion and Future Work

The objectives were to identify and evaluate existing solutions for detecting and mitigating reentrancy attacks and to propose a more effective solution that combines manual testing and static analysis.

A comprehensive review of the literature was conducted to achieve these objectives, and two tools, Hardhat and Slither, were selected for further evaluation. Hardhat was used to perform manual testing through the creation of test cases, while Slither was used for static code analysis.

The proposed solution combined both approaches, with Hardhat being used to detect reentrancy attacks through the creation of test cases, and Slither being used to further verify the presence of vulnerabilities.

The key findings of this research were that the proposed solution was able to effectively detect and mitigate reentrancy attacks in smart contracts. This was demonstrated through successfully detecting vulnerabilities in a sample smart contract using Hardhat and Slither.

The implications of this research are significant for developers of smart contracts, as it provides a reliable and efficient method for detecting and mitigating reentrancy attacks prior to deployment. This can help to prevent losses due to attacks and increase the overall security of smart contracts.

There are some limitations to this research, however. While the proposed solution was effective in detecting reentrancy attacks, it is not applicable to other types of vulnerabilities. Additionally, the sample size used in this study was small, and further testing with a larger sample size would be necessary to confirm the efficacy of the proposed solution.

There are several potential avenues for future work in this area, including the integration of dynamic analysis techniques such as fuzz testing, and the development of a commercial product based on the proposed solution. Overall, this research represents a step forward in the detection and mitigation of reentrancy attacks in smart contracts and has the potential to greatly improve the security of these important tools.

References

Alkhalifah, Ayman, N. A., Watters, P. A. ,. & ASM, 2021. A mechanism to detect and prevent Ethereum blockchain smart contract reentrancy attacks. *Frontiers in Computer Science,* 3(Frontiers Media SA), p. 598780.

Atzei, Bartoletti, N. a., Cimoli, M. a. & Tiziana, 2017. A survey of attacks on ethereum smart contracts (sok). *International conference on principles of security and trust,* pp. 164-186.

Buterin, V., 2014. A next-generation smart contract and decentralized application platform. *white paper,* Volume 3, pp. 2-1.

Chen, H., Pendleton, M., Njilla, L. & Xu, S., 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR),* Volume 53, pp. 1-43.

Chen, T. et al., 2020. SODA: A Generic Online Detection Framework for Smart Contracts.. *NDSS.*

Feist, J., Grieco, G. & Groce, A., 2019. Slither: A Static Analysis Framework for Smart Contracts. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB),* Issue 10.1109/WETSEB.2019.00008, pp. 8-15.

Huang, et al., 2017. A decentralized solution for IoT data trusted exchange based-on blockchain. *2017 3rd IEEE International Conference on Computer and Communications (ICCC),* Issue IEEE, pp. 1180-1184.

Kaleem, M., Mavridou, A. & Laszka, A., 2020. Vyper: A security comparison with solidity based on common vulnerabilities. *2020 2nd Conference on Blockchain Research \& Applications for Innovative Networks and Services (BRAINS),* Issue IEEE, pp. 107-11.

Kim & Tai-hoon, 2016. A study of digital currency cryptography for business marketing and finance security. *예술인문사회융합멀티미디어논문지,* 6(인문사회과학기술융합학회), pp. 365-376.

Koscher, K. a. C. A. a. R. F. a. P. S. a. K. T. a. C. S. et al., 2010. Experimental security analysis of a modern automobile. *2010 IEEE symposium on security and privacy,* pp. 447-462.

Luu, L. et al., 2016. Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security,* pp. 254-269.

Mense, A. & Flatscher, M., 2018. Security vulnerabilities in ethereum smart contracts. *Proceedings of the 20th international conference on information integration and web-based applications \& services,* pp. 375-380.

Morrison, R., Mazey, N. C. & Wingreen, S. C., 2020. The DAO controversy: the case for a new species of corporate governance?. *Frontiers in Blockchain,* Volume 3, p. 25.

Nguyen, T. D., Pham, L. H. & Sun, J., 2021. SGUARD: towards fixing vulnerable smart contracts automatically. *2021 IEEE Symposium on Security and Privacy (SP)},* pp. 1215-1229.

Rodler, M., Li, W., Karame, G. O. & Davi, L., 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934.*

Salehi, M., Clark, J. & Mannan, M., 2022. Not so immutable: Upgradeability of Smart Contracts on Ethereum. *arXiv preprint arXiv:2206.00716.*

Samreen, N. F. & Alalfi, M. H., 2020. Reentrancy vulnerability identification in ethereum smart contracts. *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE),* Issue IEEE, pp. 22-29.

Sayeed, S., Marco-Gisbert, H. & Caira, T., 2020. Smart contract: Attacks and protections. *IEEE Access,* Volume 8, pp. 24416-24427.

Sharma, T., Zhou, Z., Miller, A. & Wang, Y., 2022. Exploring Security Practices of Smart Contract Developers. *arXiv preprint arXiv:2204.11193.*

Sklaroff, J. M., 2017. Smart contracts and the cost of inflexibility. *U. Pa. L. Rev.,* Volume 166, p. 263.
Zhang, P., Xiao, F. & Luo, X., 2019. SolidityCheck: Quickly detecting smart contract problems through regular expressions. *arXiv preprint arXiv:1911.09425.*