5-2023

Machine Learning Based Framework for Smart Contract Vulnerability Detection in Ethereum Blockchain

Qusai Omar Mustafa Hasan
qoh3130@rit.edu

MACHINE LEARNING BASED FRAMEWORK FOR SMART CONTRACT VULNERABILITY DETECTION IN ETHEREUM BLOCKCHAIN

By

Qusai Omar Mustafa Hasan

A Thesis Submitted

in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

Computing Security

Supervised by

Dr. Wesam Almobaideen

Professor of Computing Security and Networking

Department of Electrical Engineering and Computing

Rochester Institute of Technology - Dubai Campus

United Arab Emirates

May 2023

RIT

Master of Science in

Computing Security

Thesis Approval

Machine Learning-Based Framework for Smart Contract Vulnerability Detection in Ethereum Blockchain

Student Name: Qusai Omar Mustafa Hasan

Dr. Wesam Almobaideen
Professor
Dept. of Electrical Engineering and Computing

_ _
(Thesis Advisor)

Dr. Huda Saadeh
Assistant Professor
Dept. of Electrical Engineering and Computing

_ _
(Committee Member)

Dr. Kevser Akpinar
Assistant Professor
Dept. of Electrical Engineering and Computing

_ _
(Committee Member)

Abstract

Blockchain technology is a disruptive technology that revolutionized digital payments and transactions of digital assets. Blockchain transactions operate using smart contracts which are automated software code that facilitates transactions between parties without the need for intermediary systems. Smart contracts have become an increasingly popular means of conducting transactions and executing code in a decentralized manner. As it can be written in various languages which have their flaws in terms of logic and vulnerabilities, also the immutability and autonomy of smart contracts also make them vulnerable to various security threats. Security for smart contracts is essential as exploiting bad logic or vulnerabilities in the code can lead to financial losses of digital assets as well as undermining the integrity of blockchain technology. As such, validating the security posture of smart contracts is now essential. Several static tools which can detect specific attacks on smart contracts exist. However, a comprehensive automated solution is not available.

This thesis provides a comprehensive survey of the various attack detection techniques used in smart contracts, including static analysis, dynamic analysis, and hybrid approaches. We also discuss the advantages and limitations of each approach and provide a comparative analysis of the existing tools used for the different types of smart contract analysis techniques. Furthermore, we present a machine learning based approach for the detection of attacks on smart contracts. We developed a tool that collects data from etherscan.io, which was not previously available. After collecting the dataset, static detection tools were used to test the data. The results of these tools were manually multi labeled and then fed into machine learning algorithms. The purpose of this process is to improve the accuracy of the dataset, and reduce the time cost of getting results.

Results shown for four ML models, namely Decision Tree, Perceptron, Support Vector Machine (SVM), and Long Short-Term Memory (LSTM) are used for this research based on final datasets and sub dataset and the best accuracy results for full dataset 85.7% using SVM, Reentrancy dataset 97.7% using LSTM, Etherlock dataset 80.9% using LSTM, integer overflow/underflow dataset 100% using SVM, Perceptron, and LSTM, Overall LSTM was the highest algorithm in terms of accuracy but the lowest in terms of Time cost.

Keywords: Blockchain, Smart Contract, Machine Learning, Vulnerability, Detection Tools.

Acknowledgment

Firstly, I would like to thank my supervisor, Dr. Wesam Almobaideen, for his invaluable guidance, insights, and encouragement throughout the entire thesis writing process. His wisdom and expertise have been instrumental in shaping the ideas and approach of this study.

I am indebted to my family, whose unconditional love, support, and encouragement have motivated me constantly. Their unwavering support has kept me focused and motivated throughout my academic journey.

Furthermore, I would like to thank my friends, who have been a great source of emotional support and inspiration. Their words of encouragement and advice have helped me to overcome the various challenges I faced during the course of this research.

Finally, I would like to thank again Dr.Weasm Almobaideen for his invaluable feedback and comments on earlier drafts of this thesis. His constructive criticism has helped me to refine and improve my work.

Once again, thank you all for your valuable support and encouragement throughout my academic journey.

Dedication

I dedicate this thesis to my parents, teachers, and my siblings, for their unwavering love, support, and encouragement throughout my academic journey. Their guidance and belief in me have been a constant source of inspiration, and I am grateful for their unwavering presence in my life. This thesis is also dedicated to all those who have supported me in any way, directly or indirectly, in pursuing my academic goals.

Table of Contents

List of Figures

List of Tables

Chapter 1: INTRODUCTION

Blockchain technology, one of Bitcoin's underlying technologies, has received more interest since Satoshi Nakamoto created the cryptocurrency in 2008 [1] [2] [3] [4]. The scope of blockchain applications has now expanded from digital currency [5] to all facets of life. Blockchain technology has already been proposed and deployed in numerous fields, including finance [6], IoT [7], health insurance [8], and electronic voting [9].

The foundation of these applications is smart contracts, which are coded agreements that enable individuals to comply with them without the need for trust [10]. Despite their benefits, developers may inadvertently create vulnerabilities in smart contracts through misinterpreting the code language, imperfect contract design, or carelessness. Such weaknesses are sought out by hackers and can result in significant financial losses [11].

1.1 BLOCKCHAIN OVERVIEW

This section illustrates an overview of blockchain and smart contract technologies. It also gives a categorization of blockchain technologies and then goes through different blockchain platforms that can help with smart contract development.

A consensus algorithm is composed of rules that permit various nodes in a decentralized network to come into an agreement about one truth. To guarantee agreement between all nodes about their ledgers' states within a decentralized blockchain system without any centralized authorities verifying its transactional data requires using an agreed-upon consensus algorithm. Validating transactions' accuracy to make sure they are free of errors and fraudulent activities such as double spending or other kinds of cyberattacks on blockchain networks, is ensured by a consensus algorithm. Various forms exist for consensus algorithms including PoW (Proof of Work), POS (Proof of Stake) & DPoS (Delegated Proof of Stakes) [12]. Each type comes with a unique set of merits and demerits. Without consensus algorithms, as a key component of blockchain technology, it would be impossible to maintain the security and integrity of decentralized networks [13].

1.1.1 CATEGORIES OF BLOCKCHAIN

The different categories of blockchain systems are identified, each with its governance and structure as follows [14]:

- Public blockchain [15] anybody can participate in this category of blockchain as the record is available as public. Moreover, parties can participate as members of the consensus process. In this blockchain, Immutability is high, but on the other hand, efficiency is low.
- Private blockchains [16] belong to a certain group, and only nodes from that organization are permitted to participate in the consensus process. Private blockchains are less immutable than public blockchains, but they are more efficient.
- Consortium blockchain [17] is a hybrid of the preceding two systems in which a pre-selected number of participants can join in the consensus process, even though not all users are affiliated with the same enterprise. Immutability and efficiency are similar to a private blockchain. A consortium blockchain is a middle ground between shared distributed and centralized private blockchains in terms of centralization. A permissioned blockchain is another name for this type of blockchain.

1.1.2 BLOCKCHAIN PLATFORMS

In recent years, various platforms for developing blockchain systems have been established. The following are some of the most well-known:

- Ethereum [18] Vitalik Buterin established a public, open-source blockchain platform in 2013. A decentralized platform known as Ethereum operates with its own digital currency called Ether. Smart contract creation and development are both possible on this network. As a medium of exchange within the Ethereum network and for paying transaction fees ether is employed [19]. Execution of smart contracts on the Ethereum network happens through a virtual machine known as Ethereum Virtual Machine (EVM). Being sandboxed allows for the isolation of code running on this runtime environment from other parts of the network. The EVM carries out two crucial tasks: enforcing Ethereum network rules and executing smart contract codes [20]. The focus of the Nakamoto consensus protocol utilized by Bitcoin is mainly centered around value transfer, which poses its biggest

weakness. However, Ethereum surpasses this limitation through enabling the development of decentralized applications. Moreover, Ethereum presents the concept of gas fees that serve as payment for the computational resources essential in performing smart contracts. By using ether, it will prevent a well known vulnerability known as denial of service attacks and helps to maintain network stability and security [21]

- Hyperledger Fabric [22] is a permissioned (private) blockchain platform with a modular design, smart contracts, and customizable consensus and membership services developed by IBM and Digital Asset. Hyperledger Fabric differs from Ethereum in several ways, including the lack of a built-in token and a lower amount of customization options.
- Corda [23] The R3 company created Corda blockchain technology. Corda is a business-focused distributed ledger and smart contract platform that is open-source. Corda is permissioned, and unlike Hyperledger Fabric, it does not have a native currency, but it is more tailored, addressing the needs of the financial industry.
- Quorum blockchain [24] a permissioned enterprise-grade distributed ledger technology based on Ethereum, is characterized by its unique consensus mechanism, called Istanbul BFT. The network also leverages privacy-enhancing features such as private transactions, which enable the confidential transfer of assets and shield the identities of the parties involved from the public. In addition to this point regarding its support of smart contracts it's important to mention that these are self-executing computer programs aimed at automating business processes while enforcing rules and regulations

Table 1: Blockchain Platforms Comparison

Category	Ethereum	Hyperledger Fabric	Corda	Quorum
Industry	General Purpose	Cross Industry	Financial Services	Financial Services
Mode of operation	Public blockchain	Permissioned blockchain	Permissioned blockchain	Permissioned blockchain
Consortium network support	Limited	Strong	Strong	Strong
Decentralization	Highly decentralized	Less decentralized	Less decentralized	Moderately decentralized
Consensus Protocols	Proof of Work, transitioning to Proof of Stake	Pluggable (supports multiple)	Pluggable (supports multiple)	Istanbul BFT
Transaction throughput (TPS)	15-45 TPS (varies)	Up to 20,000 TPS (varies)	Up to 200 TPS (varies)	Up to 100 TPS (varies)
Smart Contract Support	Full support	Limited (smart contract-like chaincode)	Limited (smart contract-like flows)	Full support
Smart Contract Privacy	Limited privacy (transactions are public)	Flexible privacy options (public, private, confidential)	Strong privacy features (encrypted transactions)	Flexible privacy options (public, private, confidential)
Native Cryptocurrency	Ether (ETH)	None	None	None (uses Ether as a placeholder)

1.2 SMART CONTRACT OVERVIEW

Smart contracts are electronic agreements or contracts that are self-executed based on a set of pre-defined circumstances and rules using programming instruction codes and computational infrastructure [25] [26] [27].

Smart contracts were initially proposed in 1994 by Nick Szabo and became widely employed with the advent of Blockchain [25]. This is due to the following properties [25] [27] [28].

- Absence of intermediaries: smart contract operations are carried out without the involvement of a third party.
- Automation and accuracy: smart contracts are coded with specified instances and performed automatically, removing the need for manual intervention and reducing the possibility of human error.
- Reliability and Immutability: Blockchain technology's advanced crypto methodology makes it hard to change or remove records, giving them a sense of trustworthiness.

As demonstrated in **Figure 1**, a smart contract transition is triggered by predetermined circumstances selected by individuals. The accompanying value is transferred as a transaction to the blockchain's pool, where it will be processed and confirmed by the peer-to-peer network members.

A blockchain chooses a set of transactions from the pool to be completed and validated by the blockchain's participating peer-to-peer networked nodes, and once validated, it will be added to the blockchain as a block that is linked to the previous one [25] [27].

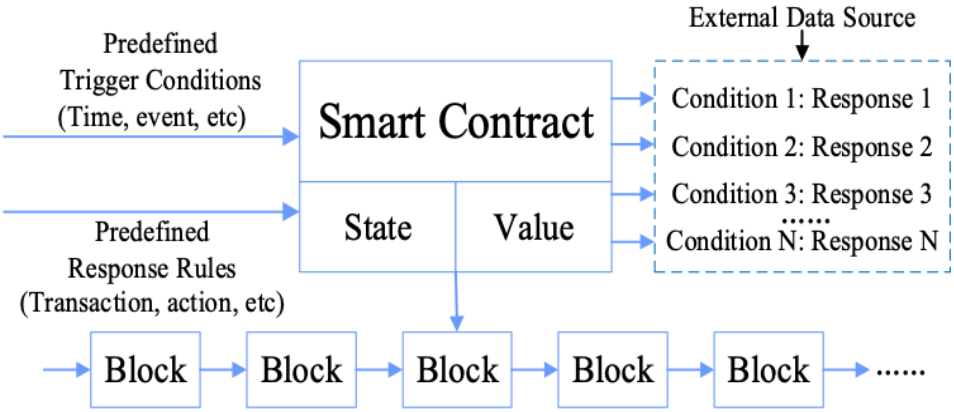

Figure 1: Smart Contract Blocks Details [28]

Many platforms, such as Ethereum and Hyperledger fabrics, can be used to develop smart contracts linked to business concerns; however, not all platforms, such as the Bitcoin blockchain [26] [29], support smart contract implementations.

1.3 MACHINE LEARNING OVERVIEW

Machine learning (ML), which has grown relatively widespread in research, has been applied to a broad variety of applications. Some examples of these applications include text mining, the detection of spam, video recommendation, image categorization, and the retrieval of multimedia information [30]. Deep learning, or DL for short, is one of various machine learning techniques, and it's widely applied in these applications [31].

Recently, modern machine learning techniques have made it possible to accurately identify cyberattacks and detect them in real-time and during post-incident investigation [32]. Notably, supervised and unsupervised machine learning techniques have been effectively applied to support intrusion detection and prevention systems, as well as to find system abuses and security breaches [33].

Anomaly-based intrusion detection systems employing ML methods are able to conform to the typical operating status of a system, isolating and identifying anomalies as unusual behavioral deviations. Due to their potential to detect zero-day attacks, i.e., assaults that exploit unknown vulnerabilities, anomaly detection techniques are hence appealing [34].

1.4 PROBLEM STATEMENT AND MOTIVATION

Nowadays the security of smart contract code cannot be guaranteed due to the complexity of the programming languages used to generate smart contracts [35]. This is now a serious and widespread problem with smart contracts, since anyone has the ability to join public Ethereum [36]. Furthermore, because smart contracts frequently manage massive amounts of financial assets, they are a main target for cyberattacks [37]. Unlike conventional programs, once smart contracts are in use, they cannot be changed to allow for the benefit of being anti-tampering. This, nevertheless, creates a serious security risk [38].

To check the vulnerability of a smart contract in Ethereum, developers traditionally submit their code for auditing which can take up to two weeks. In order to find any potential security vulnerability in the smart code a team of professionals needs to conduct a manual review. However, various tools for identifying the vulnerability of solidity smart contracts, such as Securify [39] and MAIAN [40], have been developed for users and developers who wish to inspect and assess the security of the smart contracts they are using or constructing.

The techniques employed by these tools when scanning for vulnerabilities may differ considerably from one another [41]. While some tools depend on either static [42], symbolic [43], or dynamic [44] analyses for their operations respectively. Others prefer to merge two of these methods. The supported vulnerability types for these tools may vary as well from one tool to another. The detection capabilities of various tools differ from one tool to another [45]. For example, one tool can be better suited for pinpointing particular vulnerability types such as reentrancy or overflow compared to another tool. Employing multiple techniques is necessary for a complete examination of the code despite these tools being useful in detecting initial security vulnerabilities.

The severity of this security risk has been demonstrated by numerous actual incidents. For instance, 3.6 million Ether were stolen via a weakness known as DAO in the code of [46]. Before being used, contracts must be properly examined, made leak-proof, and made sturdy.

1.5 RESEARCH AIM AND OBJECTIVE

The aim of this thesis is to detect vulnerabilities in smart contracts using a combination of static analysis tools and machine learning techniques. The following figure gives an overview of our methodology.

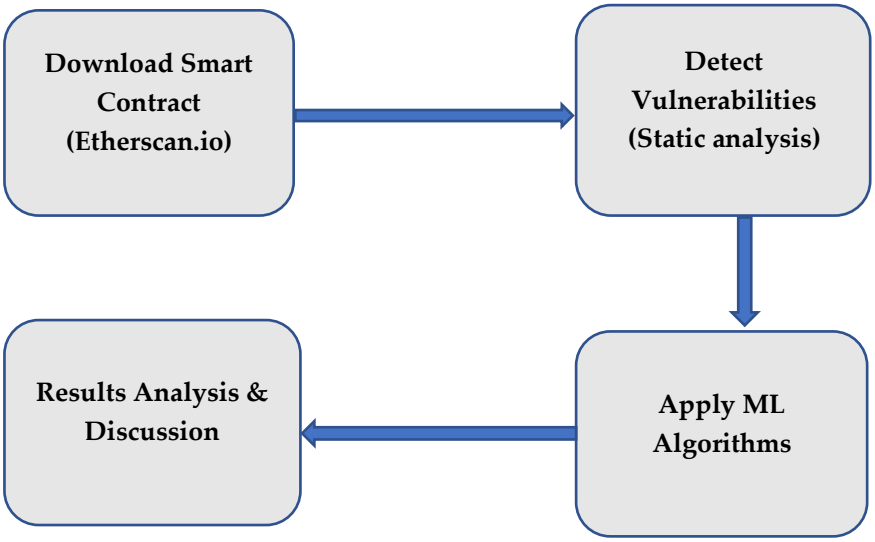

Figure 2: Summarized Methodology.

The following points highlight the contribution we have achieved in this thesis:

- Providing an overview of the existing techniques, tools, and methods for detecting vulnerabilities in smart contracts by identifying and evaluating the various smart counteract security tools available.
- Analyzing and summarizing the different detection tools used for smart contract security analysis, based on their detection methods and supported vulnerability types.
- Presenting machine learning solutions for detecting various vulnerability categories, this reduces the time, and effort and increased the accuracy.
- Proposing and implementing an automated tool that can be used to collect smart contracts, bytecode, and opcode from etherscan.io.
- Generating a dataset with seven different vulnerabilities that can be used to detect vulnerabilities related to smart contracts and avoid the overhead and inefficiency of static tools.
- Generating three sub-datasets for three different vulnerabilities, Etherlock, integer overflow/underflow, and reentrancy from the main dataset by taking the

intersections between different static tools results. These datasets can be used to detect specific vulnerability categories.

1.6 ORGANIZATION OF THE THESIS

The Thesis consists of five chapters and is organized as follows. The next chapter explains and presents a comprehensive and detailed background and Literature review of smart contract vulnerabilities, detection methods, and machine learning methods. In Chapter 3, Research Methodology will be explained in details. Chapter 4 summarizes the significant findings and results. Lastly, we conclude the Thesis and present the Future work in Chapter 5.

Chapter 2: BACKGROUND AND LITERATURE VIEW

2.1 SMART CONTRACT VULNERABILITY

Emerging today are numerous varieties of smart contract vulnerabilities. The causes of this are closely related to the blockchain platform's functionality, developer code writing, and contract design [46]. There are many reasons why smart contracts are vulnerable to security attacks [47].

(1) Due to the complexity of the code required to create smart contracts, which can lead to mistakes and errors.
(2) The second reason is that smart contracts operate in a decentralized and trustless environment, which means that there is no central authority to oversee and regulate the contracts.
(3) Smart contracts are often used to facilitate financial transactions and other valuable exchanges, which makes them a main target for hackers looking to exploit any weaknesses or vulnerabilities.

Therefore, detecting smart contracts vulnerabilities prior to deployment can increase the security of smart contracts after deployment. Since smart contracts' code is manually created, there is a strong likelihood that there may be mistakes or flaws that allow hackers to exploit them.

In order to detect smart contract vulnerabilities, this thesis investigates the following categories of vulnerabilities.

1. Arbitrary_Memory_Access:

Arbitrary Memory Access (AMA) [48] is a term used in Solidity to describe the capability of accessing any place in memory without being subject to bounds checking or other limitations. This indicates that an attacker might replace data in memory without permission to edit. This could result in security vulnerabilities and the possibility of exploits being used. For instance:

```
function vulnerableFunction(uint[] memory data) public {
  uint x = data[10];
  uint y = data[11];
  uint z = data[x + y];
}
```

Figure 3: Arbitrary Memory Access sample vulnerability.

The function in this code gets information from one array and utilizes it to generate an index for another array. The computation could, however, result in an out-of-bounds access if the input data is designed in a specific way. An attacker could then use this access to read or change memory beyond of the intended range [55].

2. Assertion_Failure:

Assertion_Failure in Solidity refers to a scenario in which an adversary purposefully brings about an assertion failure in a smart contract in order to control its behavior to cause an effect that is unanticipated. An assertion failure happens in Solidity, as indicated earlier when a condition that is anticipated to be true is discovered to be untrue during the execution of a contract. This may happen for several reasons such as having wrong inputs, making inaccurate assumptions, or having logical problems in the contract code. If the contract does not properly handle the assertion failure and does not revert its state, it may continue to execute in an unexpected or insecure way. Consequently, an attacker might be able to gain unauthorized access, modify contract data, or steal funds [49]. For instance:

```
function vulnerableFunction(uint x) public {
  assert(x > 0);
  // do something
}
```

Figure 4: Assertion Failure Sample Vulnerability.

In this code, the function utilizes an assertion to enforce the assumption that the input argument x is higher than zero. However, the assertion will fail and the function will end prematurely if an attacker supplies a value of x that is zero or negative [56].

3. Block_Dependency:

Block Dependency occurs when Ethereum smart contracts are unable to make direct calls to the built-in functions included inside smart contracts to produce a random integer. Because of this, programmers typically use the block parameters as the fundamental seeds

when implementing a random number generation function. Some examples of block parameters include the block number (block.number), the block timestamp (block.timestamp), the block hash (block.blockhash), and other related block parameters. However, in a manner analogous to the dependency on the timestamp, attackers can manipulate the block parameters in advance. This results in the generated random number being predictable, which can be exploited by malicious attackers to produce random numbers that are advantageous to the attackers themselves [50]. For instance:

```
1   function vulnerableFunction() public {
2     uint blockNumber = block.number;
3     // do something
4     require(block.number > blockNumber);
5   }
```

Figure 5: Block Dependency Sample Vulnerability.

The function in this code verifies that the block number at the moment of execution is greater than the block number at the beginning of the function. However, if a hacker delays the function's execution by altering the blockchain, the block number may stay the same or even go down, making the attack successful [57].

4. Etherlock:

The EtherLock vulnerability is a type of exploit that targets smart contracts on the Ethereum blockchain. It involves the creation of a malicious contract that appears to mimic the functionality of a legitimate contract, but with the added ability to freeze or lock up the funds within it. The attacker can manipulate the transaction flow or the user interface to force the victim to use the malicious contract instead of the legitimate one. Once the victim's funds are inside the malicious contract the attacker can trigger a function within the contract that freezes or locks up the funds. This can make the funds inaccessible to the victim. One example of an EtherLock vulnerability is when a smart contract accepts Ether payments but fails to include instructions for sending that Ether back out of the contract, such as using the "send", "call", or "transfer" methods. If these instructions are not present or are not accessible then the Ether received by the contract becomes stuck inside it indefinitely [51]. For example:

```
bool locked;

function vulnerableFunction() public {
  require(!locked);
  locked = true;
  // do something
  locked = false;
}
```

Figure 6: Etherlock Sample Vulnerability.

In order to protect against reentrancy attacks, the method in this code employs a boolean variable called locked. The lock mechanism will malfunction, though, and the attacker will be able to run arbitrary code if they are able to call the function once more before the value is reset to false [58].

5. Integer overflow/underflow:

There is a possibility of integer overflow or underflow attacks in Solidity smart contracts. When a variable value is greater than or less than the range permitted by its data type, this occurs. A variable will wrap around to 0 and begin counting from there, for instance, if its value is raised to 256 although it can only store values between 0 and 255. Attackers who purposefully input values that are larger or smaller than the permitted range can take advantage of these flaws. Attackers can use this to cause the contract to behave in unintended ways, such as transferring more funds than intended or executing unexpected operations. To prevent these attacks, developers should carefully choose the appropriate data types and check for boundary conditions when implementing mathematical operations. Additionally, input validation and access control mechanisms can be used to prevent malicious input values from being processed by the contract [52]. For instance:

```
function vulnerableFunction(uint x) public {
  uint y = x * 2;
  require(y > x);
  // do something
}
```

Figure 7: Integer Overflow/Underflow Sample Vulnerability.

The function in the above code multiplies the input argument x by 2, which may result in an overflow if x is close to the uint256's maximum value. The variable y will wrap around

to zero if the overflow happens, which could lead to the next check passing even though y is actually smaller than x [59].

6. Reentrancy:

Reentrancy is another vulnerability that can occur in Solidity smart contracts. If a smart contract is designed to do multiple things in response to one transaction, a reentrancy attack can occur when a malicious attacker interrupts the process and enters the contract again with new instructions before the current process is complete. This can cause unexpected behaviors in the contract and potentially allow the attacker to exploit it to their advantage. Attackers can exploit this vulnerability to repeatedly call certain functions, causing them to execute multiple times and potentially steal funds from the contract. To prevent reentrancy attacks, developers can use safeguards such as limiting the amount of gas available to external calls or using the "check effects interaction" pattern to ensure that all state changes are completed before any external calls are made. Proper testing and auditing of the contract code can also help identify and address any potential reentrancy vulnerabilities [53]. For instance:

```
contract SafeContract {
  mapping(address => uint256) public balances;

  function withdraw() public {
    uint256 amount = balances[msg.sender];
    require(amount > 0, "Insufficient balance");
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
    balances[msg.sender] = 0;
  }
}

contract AttackContract {
  SafeContract private targetContract;

  function setTargetContract(address _target) public {
    targetContract = SafeContract(_target);
  }

  function attack() public payable {
    require(msg.value > 0, "Invalid amount");
    targetContract.withdraw();
  }

  fallback() external payable {
    if (address(targetContract).balance > 0) {
      targetContract.withdraw();
    }
  }
}
```

Figure 8: Reentrancy Sample Vulnerability.

Users may withdraw money from the SafeContract according to this code. The withdraw function, which assumes that the contract won't call back into the function before it's finished, sends the entire balance of the calling address to the caller.

But the AttackContract can take advantage of this weakness by repeatedly calling the withdraw method before it has finished, then using the fallback function to enter the withdraw procedure again and empty the contract's money [60].

7. TODAmount (race condition):

Transaction Ordering Dependency (TOD) vulnerability is a critical security issue that can occur in smart contracts written in Solidity, the programming language used for creating Ethereum blockchain-based applications. The vulnerability arises when a contract's behavior or outcome is dependent on the order in which transactions are processed by the network. This means that an attacker can manipulate the order of transactions in their favor, leading to unexpected results such as unauthorized transfers of funds or unintended contract executions. The potential for financial loss and damage to reputation due to this vulnerability highlights the importance of thoroughly testing and auditing smart contracts before deployment to ensure the security of the underlying blockchain-based systems [54]. For instance:

```
pragma solidity ^0.8.0;

contract TODVulnerability {
    uint public balance = 0;
    mapping(address => uint) public userBalances;

    function deposit() public payable {
        userBalances[msg.sender] += msg.value;
        balance += msg.value;
    }

    function withdraw(uint amount) public {
        require(amount <= userBalances[msg.sender], "Insufficient balance");
        balance -= amount;
        payable(msg.sender).transfer(amount);
        userBalances[msg.sender] -= amount;
    }
}
```

Figure 9: Transaction Ordering Dependence (TOD) Sample Vulnerability.

In this case, a user can remove a certain amount from their balance using the withdraw function. The result of the withdraw function, however, could be impacted by other ongoing transactions that change the balance or userBalances variables. The user's userBalances value might be incorrect resulting in a failed withdrawal or allowing the user to withdraw more money than they should, for instance, if another transaction deposits money into the contract after the user has called the withdraw function but before the transaction is executed [61].

2.1.1 MITIGATION OF VULNERABILITIES

Arbitrary Memory Access: By implementing sufficient array bounds checking and ensuring that only trusted contracts have access to crucial data, this risk can be reduced.

```
contract SafeContract {
  uint256[] private data;

  function updateData(uint256 index, uint256 value) public {
    require(index < data.length, "Invalid index");
    data[index] = value;
  }
}
```

Figure 10: Arbitrary Memory Access Mitigation Sample.

The updateData function in this code updates the matching element in the data array using an index and a value parameter. Before enabling the update, the required statement verifies that the index is inside boundaries [62].

Assertion Failure: By employing defensive programming strategies and double-checking each assumption the code makes, this risk can be reduced.

```
contract SafeContract {
  uint256 public value;

  function safeFunction(uint256 x) public {
    require(x > 0, "x must be greater than zero");
    // do something
  }
}
```

Figure 11: Assertion Failure Mitigation Sample.

The safeFunction function in this code performs an operation on the x parameter. In order to avoid assertion failures, the require statement verifies that x is greater than zero before letting the operation to continue [63].

Block Dependency: To reduce this vulnerability, avoid relying on block data that can be changed by miners.

```
contract SafeContract {
  uint256 public value;

  function safeFunction() public {
    uint256 currentBlock = block.number;
    // do something with currentBlock
  }
}
```

Figure 12: Block Dependency Mitigation Sample.

The block.number parameter is used by the safeFunction function in this code to obtain the current block number [64].

Etherlock: This vulnerability can be reduced by allowing users to withdraw money from a contract using the withdraw pattern.

```
contract SafeContract {
  mapping(address => uint256) public balances;

  function deposit() public payable {
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint256 amount) public {
    require(amount <= balances[msg.sender], "Insufficient balance");
    balances[msg.sender] -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
  }
}
```

Figure 13: Etherlock Mitigation Sample.

Users can deposit Ether into the contract using the deposit function in this code, and they can withdraw their money using the withdraw method. Before approving the withdrawal, the required statement verifies that the user has a sufficient balance. The call statement then sends the amount of interest to the user's address. The contract assures clients can always withdraw their money by adopting this pattern, preventing Etherlocks [65].

Integer Overflow/Underflow: Use the SafeMath library when performing operations on integers to limit the risk of an integer overflow or underflow.

```
1   import "@openzeppelin/contracts/utils/math/SafeMath.sol";
2
3   contract SafeContract {
4     using SafeMath for uint256;
5
6     uint256 public value;
7
8     function safeFunction(uint256 x) public {
9       value = value.add(x);
10    }
11  }
```

Figure 14: Integer Overflow/Underflow Mitigation Sample.

The add function from the SafeMath library is used by the safeFunction function in this code to add an x parameter to the value variable. These vulnerabilities are avoided by this function, which checks for integer overflow and underflow and cancels the transaction if necessary [66].

Reentrancy: By employing the "checks-effects-interactions" design pattern, which places any state changes before any external calls, this vulnerability can be reduced.

```
contract SafeContract {
  mapping(address => uint256) public balances;
  bool private locked;

  modifier nonReentrant() {
    require(!locked, "Reentrant call");
    locked = true;
    _;
    locked = false;
  }

  function withdraw(uint256 amount) public nonReentrant {
    require(amount <= balances[msg.sender], "Insufficient balance");
    balances[msg.sender] -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
  }
}
```

Figure 15: Reentracny Mitigation Sample.

The withdraw function in this code enables customers to remove money from the contract. By verifying that the function is not already being executed before allowing it to continue, the nonReentrant modifier protects against reentrancy attacks. The locked variable is employed to keep track of the function execution's current state. The contract guarantees that external calls are made only after all state changes have been completed by employing this pattern, eliminating reentrancy attacks [67].

TODAmount: This vulnerability can be reduced by monitoring the amount of Ethereum transmitted with a transaction using the msg.value parameter and by utilizing proper rate-limiting strategies to stop users from sending excessive amounts of Ethereum.

```
contract SafeContract {
  uint256 public totalAmount;
  uint256 public maxAmount = 100 ether;

  function deposit() public payable {
    require(totalAmount.add(msg.value) <= maxAmount, "Maximum amount exceeded");
    totalAmount = totalAmount.add(msg.value);
  }
}
```

Figure 16: TODAmount Mitigation Sample.

In this code, the maxAmount limit is checked to ensure that the total amount of Ether in the contract does not exceed it before the deposit function allows users to deposit Ether into the contract. The contract uses this method to stop users from transferring excessive amounts of Ether and triggering a TODAmount vulnerability [68].

2.2 SMART CONTRACT SOURCE CODE, BYTECODE, OPCODE

Similar to traditional software development, smart contract development employs a high-level programming language. Contracts that are self-executing are referred to as smart contracts. Currently, many platforms [69],[70], [71], and [72] facilitate the deployment of smart contracts, each with its own contract development language. The majority of these platforms, including the Ethereum blockchain platform, are developed using the Solidity programming language. In addition to Solidity, other programming languages used to develop smart contracts are Yul, JavaScript, c++, etc. [73].

Before the smart contract can be deployed on the EVM, the source code that was written for the contract has to be compiled. Compilation of a program is the process of converting source code for a program that was written in a high-level language into machine code so that the program may be executed by a computer. In a smart contract, the binary code that is utilized for execution is referred to as bytecode, or EVM code. The EVM completes the execution of the contract by compiling the bytecode into the correct opcode [74].

Creating an Application Binary Interface (ABI) as well as building a Ethereum smart contract becomes achievable by making use of a compiler [75]. and this ABI is essential for parsing out function selectors and implementing contract functions. The process requires compiling source code into bytecode first followed by interpretation it into opcode is shown in **Figure 17**.

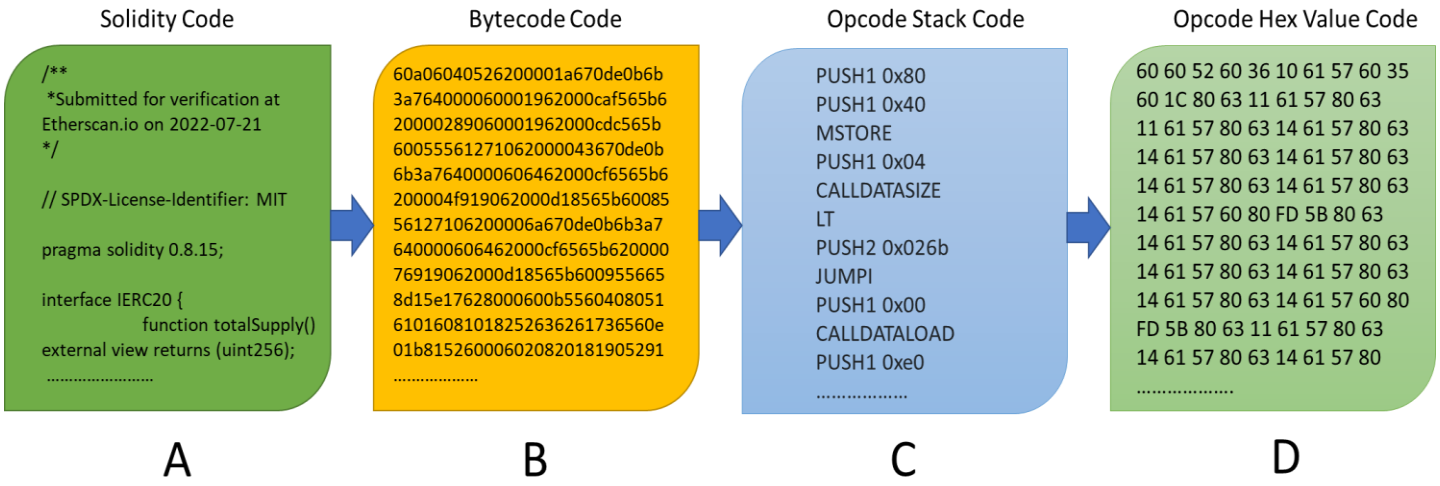

Figure 17: Solidity code, Solidity Bytecode, and Solidity Opcode.

Solidity is a high-level programming language that is used to build smart contracts on the Ethereum blockchain. An example of Solidity Source code can be found in **Figure 17** - A. These contracts have the terms of the agreement between two parties encoded straight into lines of code. When the Solidity source code is created, it has to be compiled into bytecode as shown in **Figure 17** - B, which is a low-level representation of the code that can be run by the Ethereum Virtual Machine.

The EVM does not immediately execute the bytecode; rather, it interprets the bytecode into opcode instructions as shown in **Figure 17** - C before continuing with the execution. Opcode is a low-level language that is utilized by the EVM to execute the instructions that are contained in the bytecode. Opcode is an abbreviation for "Operation Code." The EVM starts by reading each opcode from the bytecode, carrying out the operation that corresponds to it, and then moving on to the next opcode in the sequence until it reaches the end of the bytecode.

Ethereum Virtual Machine (EVM) is empowered with more than 100 distinct sets of opcode assembly operation instructions including arithmetic functions such as, add/subtract, comparison expressions such as, greater/equal/less than, bitwise functions AND/OR/XOR, cryptographic functionality (hash/encrypting etc.). Along with handy stack manipulation functionalities and **Table 2** summarized below highlights all the commonly utilized Ethereum opcode assembly instruction. The EVM uses the instruction table located at [76]. to map bytecode consisting of a series of hexadecimal numbers like 60 60 52 40 with its opcode hexadecimal value, an example, “40” maps to “MSTORE, etc..

Table 2: The most commonly opcode assembly instructions.

INSTRUCTIONS	DEFINITION
PUSH	This instruction adds value to the stack. A different byte length of the value being pushed corresponds to each of the 16 PUSH possibilities.
POP	The item at the top of the stack is removed by this instruction.
ADD and SUB	The top two items of the stack are added to and subtracted from using these instructions, respectively.
MUL and DIV	The top two items in the stack are multiplied and divided by these instructions, respectively.
SLOAD and SSTORE	These instructions, in turn, read and write data to storage.
JUMP and JUMPI	These instructions either unconditionally or conditionally jump to a new position in the bytecode depending on the item at the top of the stack.
CALL and RETURN	These instructions are put to use, respectively, to invoke external contracts and receive data from a contract.
SELFDESTRUCT	The current contract is terminated, and any remaining ether is sent to the specified address.

These are only a few of the EVM instructions that are used most frequently. There are numerous additional features, and developers who deal with Ethereum smart contracts should have an in-depth understanding of the EVM and its command set.

In this thesis we have chosen to examine the opcode level to identify vulnerabilities in smart contracts for three reasons. First, the source code is written by humans and contains function names that can be altered, which can make it difficult to identify the relevant functions and their influence on the detection results. Moreover, the source code frequently contains annotations and vacant lines, making it difficult to authentically represent the code's characteristics. Second, bytecode is not legible by humans and lacks syntax structure and sequence information, making it difficult to analyze its functions. Lastly, opcodes are derived from more than 100 EVM operation instructions, which accurately reflect the contract's inner logic and enhance the model's reliability.

2.2.1 DETECTION METHODS USED FOR VULNERABILITY DETECTION

There are four different detection techniques for smart contracts code as follows:

Static analysis is a method that can be used for examining Solidity smart contracts without executing them. This approach entails inspecting the smart contract's Solidity source code for possible vulnerabilities and ensuring compliance with coding standards. Static analysis

is very beneficial for finding code faults that might lead to vulnerabilities in Solidity smart contracts, such as integer overflow/underflow and reentrancy problems [77].

Symbolic execution is a testing approach that can also be used for Solidity smart contracts to examine the code to find all potential execution pathways. Symbolic execution represents inputs to the smart contract with symbolic values, enabling the tester to explore all potential routes of execution without actually running the code. Symbolic execution may help uncover sophisticated vulnerabilities in Solidity smart contracts, such as control flow and data flow concerns [78].

Fuzzing analysis is a method that can be used for testing Solidity smart contracts that involve creating random or semi-random inputs and analyzing the contract's behavior. Fuzzing can help as a technique for detecting vulnerabilities in Solidity smart contracts that are caused by unexpected or malicious inputs, such as buffer overflows and DOS attacks [79].

To evaluate Solidity smart contracts against unknown possible vulnerabilities, machine learning methods may be utilized. Machine learning may be used to detect patterns in Solidity source code or bytecode that indicate vulnerabilities such as code repetition or control flow concerns [80].

2.2.2 TOOLS USED FOR VULNERABILITY DETECTION

In this section, we explore the most relevant smart contract vulnerability tools that have been proposed in the literature.

1) Oyente [81]

Oyente is a symbolic execution based static analysis tool that can be used without using high-level languages like Solidity on EVM bytecode. It makes it possible to find flaws like TOD, predictable random numbers (timestamp dependent), reentrancy, and exception-handling errors. Additionally, it supports the majority of EVM opcodes. However, Oyente finds it challenging to infer the development intent simply from the EVM bytecode due to the absence of context information such as variable types and the repetition of the same bytecode by many function calls. As a result, Oyente is not capable of checking for concerns with fairness and accuracy such as integer overflow.

2) Securify [82]

Securify is a scalable and lightweight security verifier for Ethereum smart contracts. It establishes compliance patterns and compares the contract to these patterns to find weaknesses. Additionally, it offers the discovery of flaws like TOD, missing argument validation, unchecked write and transfer permissions, frozen tokens, and exception handling mistakes. Version 2.0 of Securify has recently been released and it supports an improved smart contract language and more detection techniques.

3) Mythril [83]

Mythril uses symbolic techniques [83]. Mythril initializes the contract account's state after decompiling the EVM bytecode and uses numerous transactions to explore the contract's state space. When an undesirable circumstance is discovered. Mythril determines the necessary transactions to get to a vulnerability state when one is found to confirm the vulnerability's existence.

4) teEther [84]

The focus of the analysis tool teEther [84], which uses symbolic execution and result validation to analyze Ethereum EVM contracts, focuses on identifying permission verification missing, i.e. unrestricted call. There are four steps in the teEther contract analysis process: firstly, by creating a Centrifuge token for a contract. The next step is looking in the contract for crucial instructions, such as DELEGATECALL, SELFDESTRUCT, and/or SSTORE, and other state-changing instructions. Investigating routes to these instructions is the third phase. The fourth stage consists of resolving the limits imposed by using different approaches in order to locate the weaknesses.

5) MAIAN [85]

MAIAN [85] is a tool based on symbolic execution and results in validation to analyze Ethereum EVM contracts. It checks the execution pathways as it symbolically executes smart contracts. By assaulting contracts using actual transactions. MAIAN facilitates the identification of security flaws including suicidal contracts and frozen tokens.

6) ContractFuzzer [86]

A fuzzer called ContractFuzzer is used to find vulnerabilities in Ethereum EVM contracts [86]. Online fuzzing and offline instrumentation make up the two components of ContractFuzzer. The offline instrumentation phase involves instrumenting the EVM code

so that the fuzzing phase may see the contract's execution. ContractFuzzer can extract information about ABI functions, and analyze the contract bytecode, which enables the tool to produce legitimate fuzzing inputs. The program chooses at random from among the smart contracts it has crawled on Ethereum for fuzzy processing. Then it provides the discovery of vulnerabilities like exception handling errors, reentrancy, … etc.

7) Slither [87]:

Slither [87] is a Python-based open-source static analysis framework that was created in 2018. It analyzes solidity code as input. It employs the SlithIR intermediate representation. To find vulnerabilities, Slither uses dataflow analysis and tracking methodologies. It may be used to discover automated vulnerabilities, detect automated optimization and analyze code. This tool's open-source version supports finding various issues such as re-entrancy, ether lock, and timestamp.

8) Conkas [88]:

Conkas is a tool used to detect vulnerabilities in Solidity contracts by identifying potential security issues such as re-entrancy, integer overflow, and other common vulnerabilities is done through analysing the byte code or solidity code of the smart contract. By using both dynamic and static analysis methods. It is possible to use Conkas as either an independent tool or integrate it within an existing development pipeline. Smart contract developers can find and eliminate potential issues more easily with an easy to understand report format that presents the analysis results [88].

9) The ConFuzzius [89]

ConFuzzius [89] is a tool that uses a combination of static and dynamic analysis approaches to effectively traverse the state space of smart contracts and uncover vulnerabilities like reentrancy, integer overflow/underflow, and other common flaws. The program uses data dependence analysis to discover potential vulnerability channels via the contract and prioritizes testing those paths first. The hybrid technique combines coverage-guided fuzzing with symbolic execution to improve testing efficiency and efficacy.

The authors created a dataset and published it in [90]. A DL methodology called BLSTM-ATT is used to precisely discover reentrancy issues. Their work is restricted to detecting specific attacks based on a reentrancy vulnerability and for an old version of smart contracts.

Most of the tools discussed above are not using machine learning, as these tools are based on static analysis which may not be able to detect vulnerabilities with high accuracy.

In [90] they have conducted machine learning algorithms on solidity version 0.4.X which is currently an old version as solidity has reached version 0.8.X. Therefore, the code syntax is different and consequently, there is a need to have a new dataset that represents the state-of-the-art smart contracts code.

Table 3: Literature Review Comparison Table

Tool	Detection Level		Year Published	Last Update	Platform Used	Availability on GitHub	Link of GitHub
	Solidity Source Code	Bytecode					
ContractFuzzer	✓	-	2018	2020	Python	✓	https://github.com/gongbell/ContractFuzzer [91]
Slither	✓	✓	2018	2023	Python	✓	https://github.com/crytic/slither [92]
Conkas	✓	✓	2021	2022	Rust	✓	https://github.com/nveloso/conkas [93]
Confuzzius	✓	✓	2020	2022	Python	✓	https://github.com/christoftorres/ConFuzzius [94]
MAIAN	✓	✓	2018	2021	Java	✓	https://github.com/ivicanikolicsg/MAIAN [95]
teEther	✓	-	2018	2021	Solidity	✓	https://github.com/nescio007/teether [96]
Mythril	✓	-	2017	2023	Python	✓	https://github.com/ConsenSys/mythril [97]
Oyente	✓	-	2016	2020	Python	✓	https://github.com/enzymefinance/oyente [98]
Securify	✓	-	2018	2021	Java	✓	https://github.com/eth-sri/securify2 [99]

Table 3 shows that every tool listed is intended to verify Solidity code for weaknesses at the source code level. At the bytecode level, however, only four out of the 9 listed tools are capable of detecting vulnerabilities. The majority of the tools mentioned in **Table 3** were introduced in 2018, see year published column, but they have received frequent updates for

better performance and capabilities which is worth mentioning. A closer look reveals that a large number of these tools were updated not too long ago as illustrated by the last update column. All the tools are available on GitHub and freely accessible. Four different programming languages, Python, Rus, Java, and Solidity, have been used to create the platforms that support vulnerability detection. Among these programming languages Python was the most widely used.

Table 4: Category of Vulnerability Tools that Can Detect

Tools	Arbitrary_Memory_Access	Assertion_Failure	Block_Dependency	Etherlock	Integer overflow/underflow	Reentrancy	TODAmount
ContractFuzzer	-	-	-	-	✓	✓	-
Slither	-	-	-	✓	✓	✓	✓
Conkas	-	-	-	-	✓	✓	-
Confuzzius	✓	✓	✓	-	✓	-	✓
MAIAN	-	-	-	✓	-	✓	-
teEther	-	-	-	-	✓	✓	✓
Mythril	-	-	-	-	✓	✓	✓
Oyente	-	-	-	-	-	✓	✓
Securify	-	-	-	-	✓	✓	✓
In [90]	-	-	-	-	-	✓	-

After analyzing **Table 4**, it is evident that four out of the eight vulnerabilities are being aimed for detection by almost all the tools being considered. These vulnerabilities comprise Time-Dependent Access Control (TOD attack), Reentrancy, and Integer Over/Underflow. The unanimous agreement among the tools on the identification of these four vulnerabilities indicates that they are extremely crucial concerns in Solidity programming that necessitate vigilant consideration from developers.

On the other hand, there are four attacks that are only detected by one or two tools, implying that they may be relatively uncommon or difficult to detect. These vulnerabilities are Arbitrary Memory Access, Assertion Failure, Block Dependency, and Etherlock. It is worth noting that just because these vulnerabilities are only detected by a limited number of tools, it does not mean they are less important or less risky than the more commonly detected vulnerabilities. In fact, these less frequently detected vulnerabilities may be even more dangerous precisely because they are less well-known and less likely to be protected against. As such, a recommendation can be given that encourages researchers to conduct more detailed research that studies the risk of these least commonly detected vulnerabilities using different tools and techniques.

Table 5: Detection Methods of Each Tool.

Tools	Detection Method			
	Static	**Symbolic execution**	**Fuzzing**	**ML**
ContractFuzzer	-	-	✓	-
Slither	✓		✓	-
Conkas	✓	-		-
Confuzzius	✓	-	✓	-
MAIAN	✓	-		-
teEther	-	-	✓	-
Mythril	-	✓	-	-
Oyente	✓	-	-	-
Securify	✓	-	-	-
In [90]	-	-	-	✓

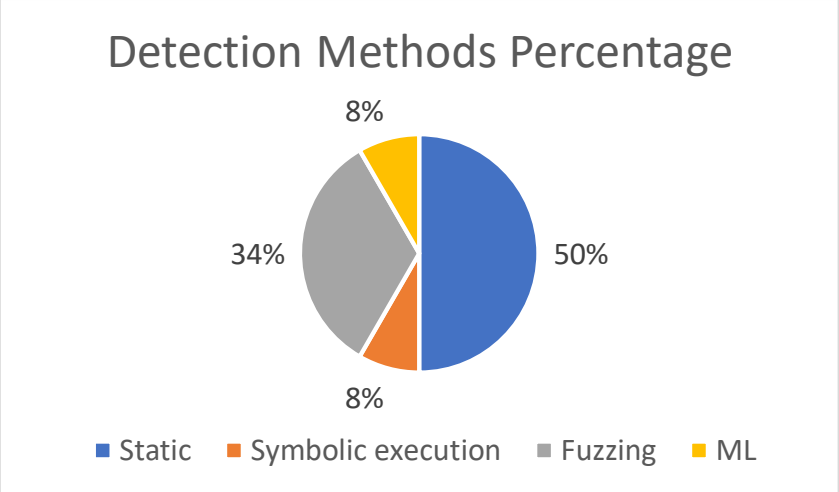

Figure 18: Statistics of Detection Techniques.

Upon careful examination of **Table 4** and **Figure 18**, it becomes evident that machine learning algorithms have been applied only to the detection of one attack category - reentrancy. While this may represent an important step forward in the use of machine learning for vulnerability detection, it also highlights the vast potential of such a tool for future research in this area.

Given that there are numerous vulnerability categories that Solidity code may be susceptible to, there is ample scope for machine learning algorithms to be applied to detect vulnerabilities beyond reentrancy. By leveraging the power of artificial intelligence and data-driven analysis, it may be possible to develop more accurate and efficient methods for identifying and mitigating a wider range of vulnerabilities in Solidity code.

Consequently, the fact that machine learning algorithms have been successfully applied to detecting reentrancy represents a powerful motivation for future work in this area. By continuing to explore the potential of machine learning in the context of Solidity vulnerability detection, researchers and developers may be able to unlock even greater potential for securing smart contracts and other decentralized applications.

In **Table 5** and **Figure 18** we can notice that only one tool is using symbolic execution technique and one tool uses a machine learning technique and the rest are either fuzzy or static. This encourages future research focusing on detecting more vulnerable categories using symbolic and/or machine learning.

2.2.1 MACHINE LEARNING MODELS

Machine Learning (ML) algorithms can improve and learn over time by analyzing substantial amounts of data, observing patterns and trends, and employing the findings to predict outcomes or make decisions. Adapting ML behavior based on received data is possible because of their learning experience. ML algorithms are classified into three primary categories, supervised learning, unsupervised learning, and reinforcement learning. Each category has its own distinct merits and demerits. The usage of machine learning algorithms can be found in different applications such as detecting malware [100], image recognition [101], blockchain [102], …etc. We will introduce four commonly used machine learning algorithms: Decision Tree (DT), Neural Network (NN), Support Vector Machines (SVM), and Long Short Term Memory (LSTM).

1. **Decision Tree:**

A decision tree is constructed by adding nodes for each feature in the pre-processed dataset, with the characteristic deemed to be the most significant being placed at the tree's root. It functions properly for both continuous and categorical output variables for each feature. This procedure is repeated until the final (leaf) node, which contains the DT's predictions or outcomes, is reached. **Figure 19** shows a prototype decision tree model. The point that has a light green color indicates where the root node is, and the red dots indicate where the leaf nodes are. There is a decision to be made at every fork in the road [103]. The decision Tree Algorithm is a popular machine learning algorithm that is used for both classification and regression tasks. One of the classes within the Decision Tree Algorithm is the Decision Tree Classifier, which is used specifically for classification tasks.

To use the Decision Tree Classifier, you first need to train the algorithm using training data. This is done by calling the "fit" method of the Decision Tree Classifier class and passing in the training data as an argument. The "fit" method then creates a decision tree based on the training data, where each node of the tree represents a decision based on a specific feature or attribute of the data. The tree is constructed by recursively partitioning the data into subsets based on the selected features until a stopping criterion is met.

Once the tree has been constructed, the Decision Tree Classifier can be used to predict the class labels of new, unseen data points. This is done by traversing the decision tree from the root node to a leaf node based on the values of the features of the new data point. The leaf node reached by this traversal represents the predicted class label for the new data point [104].

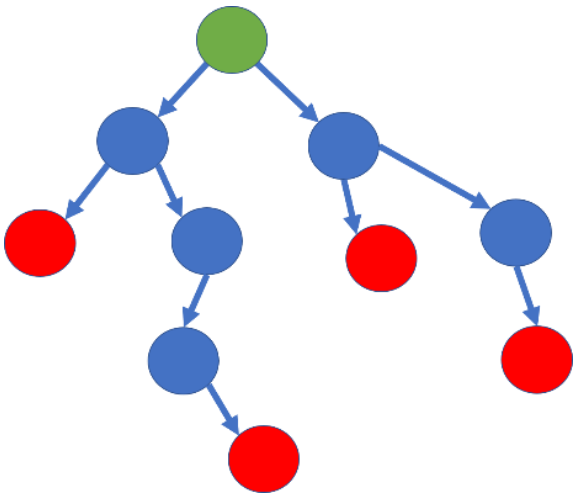

Figure 19: Decision Tree Model

2. **Neural Network:**

A neural network is a type of machine learning algorithm that is modeled after the structure and function of the human brain. It consists of a series of interconnected nodes or "neurons" that process and transmit information.

One of the simplest neural network architectures is the Perceptron, which consists of a single layer of neurons that take in input features and produce output values. The Perceptron is commonly used for binary classification tasks, where the goal is to predict whether a given input belongs to one of two classes.

While the Perceptron can be effective for simple classification tasks, it has some limitations. For example, it can only learn linear decision boundaries, which can limit its performance on more complex tasks. To address this limitation, neural networks with hidden layers are used.

A hidden layer is a layer of neurons between the input and output layers of a neural network. Each neuron in the hidden layer receives inputs from the neurons in the previous layer and produces outputs that are fed into the neurons in the next layer. By adding hidden layers,

neural networks can learn more complex patterns in the data and produce more accurate predictions. [103].

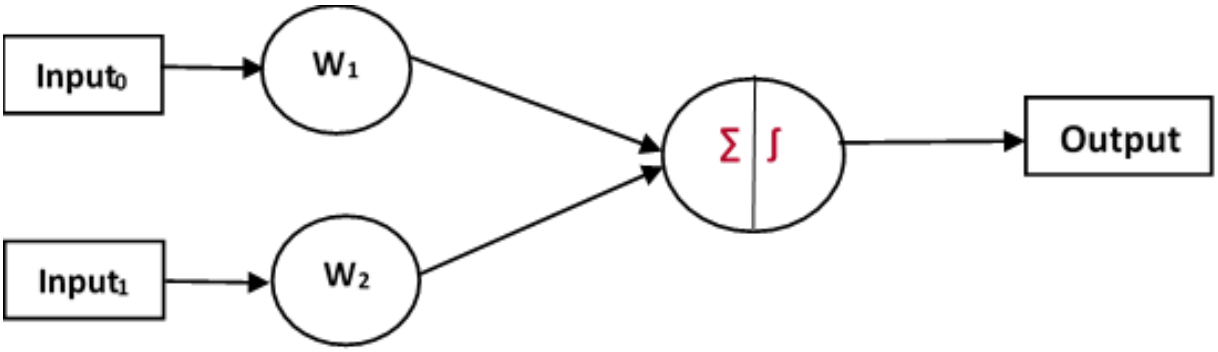

Figure 20: Neural Network

3. Support Vector Machine (SVM):

The Support Vector Machine, or SVM, is a method of supervised machine learning that may be used for regression, classification, and the identification of outliers. It provides a high level of accuracy where the numerous continuous, unconditional, and discrete variables can be handled, outperforming other classifiers like logistic regression and DT. Finding the marginal hyperplane with the highest margin that can be used for the classification of input data that has been pre-processed is the main goal of support vector machines [105]. As can be seen in **Figure 21**, it picks the Support Vector margin that has the highest probability of being correct within the dataset that is provided.

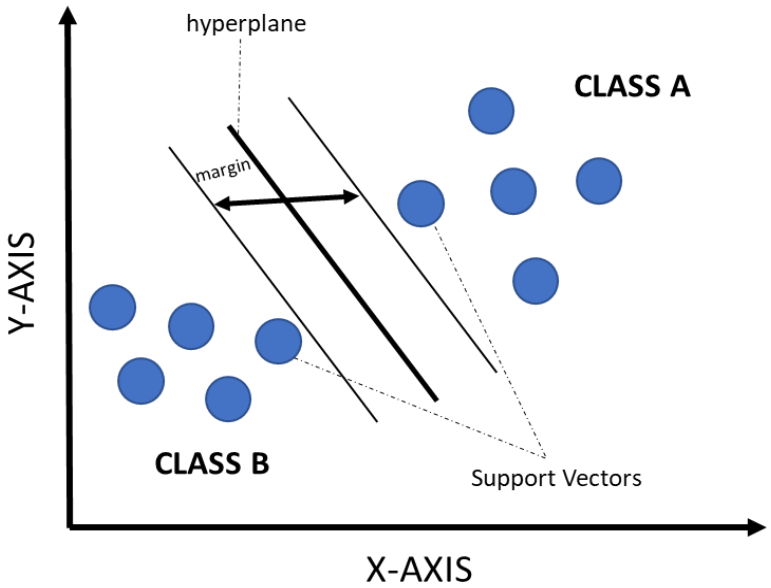

Figure 21: SVM Classifier

4. LSTM:

Long Short-Term Memory (LSTM) is a form of recurrent neural network (RNN) that is extensively employed in machine learning for modeling and predicting sequences. Traditional RNNs have difficulty preserving long-term dependencies in sequential data due to the vanishing gradient problem; LSTMs are designed to surmount this limitation. LSTMs employ a memory cell that enables them to retain and retrieve information over extended periods, thereby allowing them to model intricate temporal patterns [106].

The memory cell is accompanied by three gates in an LSTM architecture: the input gate, neglect gate, and output gate. These gates permit the LSTM to selectively store or discard information in the memory cell, thereby enhancing the model's capacity to learn and remember meaningful long-term dependencies. The input gate regulates the passage of new input into the memory cell, the neglect gate controls the removal of information from the cell, and the output gate controls the quantity of information that is output from the cell [106].

LSTMs have proven highly effective in a variety of applications, including speech recognition, machine translation, sentiment analysis, and time-series forecasting. They are a popular choice for applications such as natural language processing, financial forecasting, and weather forecasting because they are particularly well-suited to tasks where long-term dependencies are crucial for accurate predictions [106].

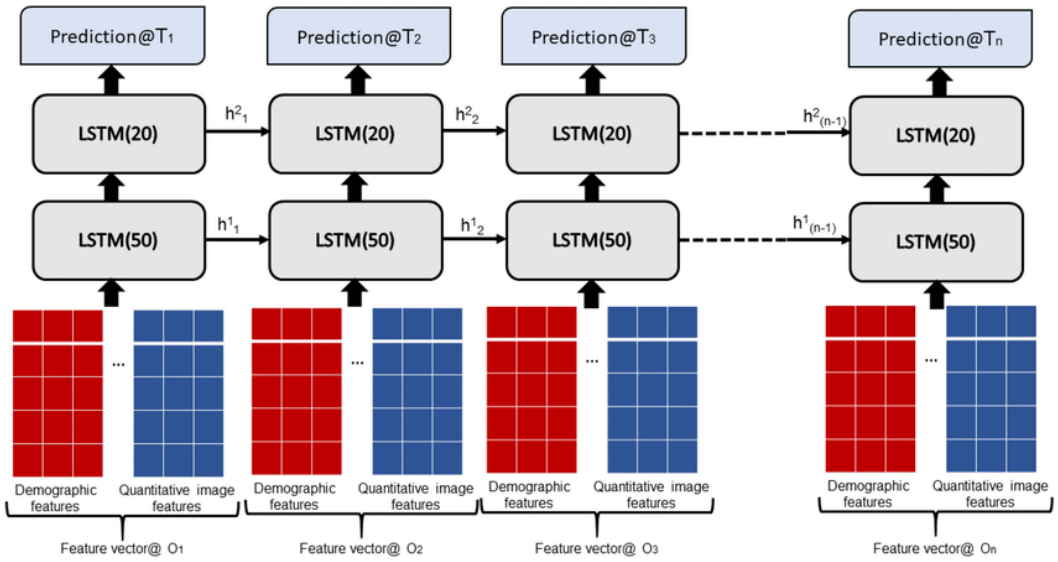

Figure 22: LTSM Classifier

2.2.2 MACHINE LEARNING VULNERABILITY DETECTION MODELS USING OPCODE

Detecting malware and vulnerabilities is crucial for maintaining the security of computer systems and networks. However, traditional approaches to malware detection and vulnerability analysis are often time-consuming and resource-intensive [107].

Machine learning has emerged as a promising approach to addressing these challenges, particularly when it comes to detecting malware and vulnerabilities at the opcode level. By analyzing the opcode sequences of executable files, machine learning algorithms can identify patterns and behaviors that are indicative of malware or vulnerability exploitation [108].

One advantage of using machine learning for malware and vulnerability detection is that it can quickly analyze large amounts of data and identify potential threats in real-time. In addition, machine learning can adapt to new types of malware and vulnerabilities, making it a more robust and scalable approach to security [107]. We will introduce four research papers that use NN, SVM, Decision Tree, and LSTM on different technologies to detect malwares based on the opcode level.

1. **Neural Network (NN):**

In [109], the Perceptron is trained on opcode features extracted from malware samples, which represent the low-level instructions executed by a program. The results of the experiment show that the Perceptron achieves good accuracy in detecting malware using opcode features, with an accuracy rate of around 90%. However, the performance of the Perceptron is inferior to that of the proposed hybrid attention network, which achieves an accuracy rate of over 98%.

The authors suggest that the Perceptron can serve as a baseline method for malware detection using opcode features, especially in scenarios where the dataset is small or the computational resources are limited. The study also highlights the potential of machine learning algorithms, including the Perceptron, for effective detection of malware.

2. SVM

In [110] the effectiveness of the SVM algorithm in detecting malware using opcode trigram sequences is evaluated in detail. Support Vector Machine (SVM) is a popular machine learning algorithm that is commonly used in binary classification tasks, including malware detection. SVM works by finding an optimal hyperplane that separates the data points into two classes, in this case, malware and benign files. In this study, the authors extract opcode trigram sequences from malware samples, which represent a sequence of three opcode instructions, and use them as input features for the SVM model.

To evaluate the effectiveness of the proposed approach, the authors conducted experiments on two publicly available datasets, namely, the Malimg dataset and the Microsoft Malware Classification Challenge (MS-Malware) dataset. The results show that the SVM algorithm achieves high accuracy in detecting malware using opcode trigram sequences, with an accuracy rate of around 99%. The precision, recall, and F1-score metrics are also reported to be high, indicating the effectiveness of the SVM approach in detecting malware.

The authors also compared the performance of the SVM algorithm with that of other machine learning algorithms, such as Naive Bayes, Random Forest, and Multilayer Perceptron. The results show that the SVM algorithm outperforms these algorithms in terms of accuracy and F1-score metrics which highlight its effectiveness in detecting malware using opcode trigram sequences.

The study also highlights the importance of feature selection in malware detection. By using opcode trigram sequences as input features, the SVM algorithm is able to effectively distinguish between malware and benign files.

3. Decision Tree

In [111] the authors conduct an in-depth evaluation of the effectiveness of decision tree algorithms in detecting metamorphic malware using opcode frequency rates. The study focuses on metamorphic malware, which is a type of malware that can change its code to avoid detection by traditional signature-based antivirus systems. To detect such malware, the authors extract opcode frequency rates from the malware samples, which represent the relative frequency of each opcode instruction, and use them as input features for the decision tree algorithm.

In the context of malware detection, the decision tree algorithm can effectively distinguish between malicious and benign files by identifying the most discriminative features that distinguish them. To evaluate the effectiveness of the proposed approach, the authors conducted experiments on a publicly available dataset, namely, the VXHeaven dataset. The results show that the decision tree algorithm achieves high accuracy in detecting metamorphic malware using opcode frequency rates, with an accuracy rate of around 99%. The precision, recall, and F1-score metrics are also reported to be high, indicating the effectiveness of the decision tree approach in detecting metamorphic malware.

The study also compared the performance of the decision tree algorithm with that of other machine learning algorithms, such as Naive Bayes, K-Nearest Neighbors, and Random Forest. The results show that the decision tree algorithm outperforms these algorithms in terms of accuracy and F1-score metrics, highlighting its effectiveness in detecting metamorphic malware using opcode frequency rates.

The authors also conducted experiments to evaluate the impact of feature selection on the performance of the decision tree algorithm. The results show that the decision tree algorithm can achieve high accuracy in detecting metamorphic malware using a small subset of the most informative features, which can significantly reduce the computational cost of the detection process.

4. LSTM

In [100] the authors propose a novel approach for malware detection using LSTM neural networks. The study focuses on the use of opcode sequences as input features for the LSTM network. Opcode sequences are a representation of the instructions executed by a program and are commonly used in the field of malware detection to analyze the behavior of malware. In the context of malware detection, the LSTM network can be used to capture the patterns and relationships in the opcode sequences and classify malware samples accordingly.

To evaluate the effectiveness of the proposed approach, the authors conduct experiments on a publicly available dataset, namely, the Malimg dataset. The dataset contains a large number of malware samples, each labeled with the corresponding malware category.

The results of the experiment show that the LSTM-based approach achieves high accuracy in detecting malware using opcode sequences as input features, with an accuracy rate of

over 98%. The precision, recall, and F1-score metrics are also reported to be high, indicating the effectiveness of the LSTM approach in detecting malware.

The performance of the LSTM based technique is also contrasted with those of other machine learning algorithms, including SVM and Random Forest. The results show that the LSTM based approach outperforms these algorithms in terms of accuracy and F1-score metrics, highlighting its effectiveness in detecting malware using opcode sequences.

The length of the opcode sequences and the number of LSTM layers are two other variables that the authors test to see how they affect the performance of the LSTM based approach. The results show that the LSTM based approach is robust to variations in the length of the opcode sequences and can achieve high accuracy with a small number of LSTM layers.

Chapter 3: RESEARCH METHODOLOGY

3.1 METHODOLOGY

The objective of this thesis is to develop a technique that can automatically and more precisely identify multiple vulnerabilities in smart contracts, overview of this technique is shown in Figure. The intention is to overcome the limitations of conventional approaches to detecting vulnerabilities in smart contracts, which suffer from drawbacks such as inadequate detection capabilities, limited automation, and slow detection speeds as shown in **Figure 32** [40].

In this thesis a framework for detecting vulnerabilities with multiple labels is proposed which consists of five parts:

1. Collecting Solidity smart contracts, their bytecode, and opcode.
2. Running solidity static detection tools to get vulnerabilities categories
3. Multi-labelling the dataset based on the results of step 2.
4. Opcode pre-processing to convert the opcode into opcode EVM hexadecimal representation.
5. Run ML vulnerability detection models on the dataset.

The first step in data pre-processing is to use the Ethereum website to convert the smart contract bytecode into opcodes. This is followed by converting the opcodes into opcode EVM hexadecimal values for input to the Machine learning detection tool.

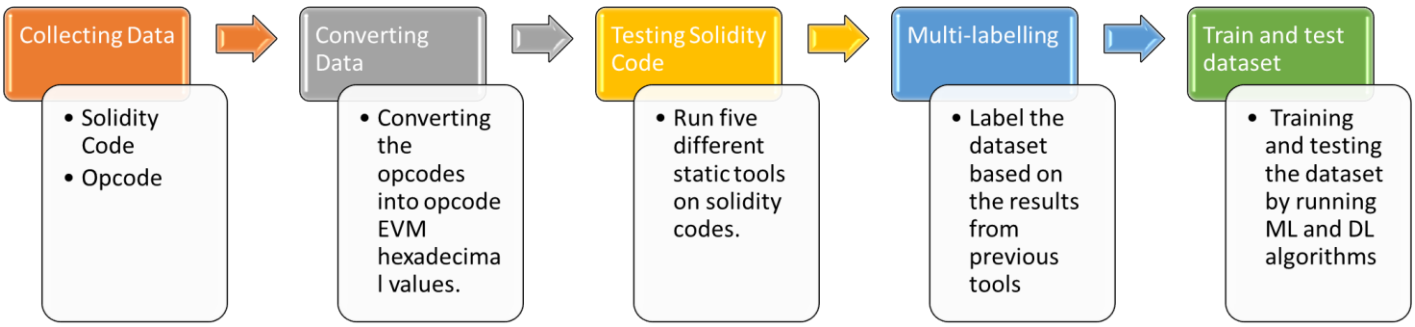

Figure 23: Our Technique Methodology.

The dataset has been labeled manually by entering the value either “0” or “1” for each vulnerability category based on the static tools results. This results in having a multi label category that can be used by machine learning algorithms for training and testing. We have used in our study four different machine and deep learning techniques (Decision Tree, Perceptron, SVM, and LTSM). We delve into the details of these five steps as follows.

3.1.1 DATA COLLECTION

Researchers and developers who need access to a large collection of smart contracts for analysis and experimentation face a challenge due to the limited availability of open source Solidity code.

In our case, we discovered that the available resources for accumulating Solidity code were insufficient to meet our requirements for a diverse and exhaustive collection of contracts. As a result, we decided to create our own Solidity code collection utility. Our tool is used to extract contracts “Solidity, bytecode, and opcode” from the official platform of Ethereum “ethersscan.io”, by using a combination of web crawling libraries in Python.

The creation of our own tool enabled us to collect a large and diverse set of contracts for analysis. This in turn allowed us to acquire a deeper understanding of the behavior of smart contracts on the Ethereum network. By using this tool, we were able to surmount the limitations of existing resources and contribute to the research community by providing an exhaustive dataset of Solidity code for analysis and experimentation. The Pseudo Code of collecting solidity code be found in Appendix A.

On the other hand, we aimed to extract the relevant opcode and bytecode for each Solidity contract once we had gathered a huge and diversified collection of Solidity smart contracts. This was a critical stage in our investigation since the opcode and bytecode contain critical information about the contract's internal workings, including its functionality and any existing flaws. We used etherscan.io again to derive the opcode and bytecode of collected smart contracts.

We utilized the opcode for each contract as input data for our vulnerability detection algorithm after obtaining them. By examining the opcode, our model was able to discover possible vulnerabilities in the contracts, such as reentrancy, integer underflow, … etc.

Collecting opcode was an important step in our investigation of smart contract vulnerabilities, allowing us to create a thorough and effective vulnerability detection

model. The Pseudo Code of collecting bytecode and Opcode code be found in Appendix A.

3.1.2 DATA PREPROCESSING

After we have extracted the opcode from the Solidity contracts, we need to convert the opcode instructions into a machine readable format. This involved converting each opcode instruction into its corresponding hexadecimal value, which could be read and processed by our vulnerability detection model.

To achieve this, we developed a custom script that maps each opcode instruction into its corresponding hexadecimal value. This script allowed us to easily convert the opcode instructions into a machine readable format, which we could then use as input data for our vulnerability detection model.

By converting the opcode instructions to their hexadecimal values, we were able to accurately represent the inner logic of the Solidity contracts in a machine-readable format. This allows our model to analyze the contracts and identify potential vulnerabilities. This step was crucial in our analysis, as it enabled us to process the opcode instructions in a standardized and efficient manner and improve the accuracy and effectiveness of our vulnerability detection model. The Pseudo Code of converting Opcode to Opcode Hexadecimal Values can be found in Appendix A.

3.1.3 DETECTING SMART CONTRACTS VULNERABILITIES USING STATIC TOOLS

Once we had collected a diverse set of Solidity smart contracts and extracted their corresponding opcode and bytecode, we used static analysis tools to detect potential vulnerabilities in the contracts.

We used a variety of static analysis tools such as, Slither [87], Confuzzion [89], MAIAN [85], Conkas [88], and Securify [82], where these tools are available on GitHub as publicly available open source tools [92], [94], [95], [93], and [99], respectively. These tools enabled us to identify a wide range of potential vulnerabilities, including reentrancy attacks, integer overflows, and other common smart contract vulnerabilities.

By using static analysis tools, we were able to identify potential vulnerabilities in a large number of contracts, and gain insights into the most common types of vulnerabilities and their prevalence in real-world contracts. This information was valuable to conduct the multi labeling process in the next step.

3.1.4 MULTI-LABEL

Table 6 displays the definition of smart contract multi-label classification, which involves categorizing smart contracts into multiple categories or labels based on static detection tools results, as follow:

Table 6: Dataset Categories.

FinalDatasetAll.csv							
Vulnerability Types							Category
Arbitrary_Memory_Access	Assertion_Failure	Block_Dependency	etherlock	integer overflow/underflow	reentrancy	TODAmount	Category
0	0	0	0	0	0	0	0 0 0 0 0 0 0
0	0	0	1	0	0	0	0 0 0 1 0 0 0
0	0	0	0	1	0	0	0 0 0 0 1 0 0
1	1	0	0	1	0	0	1 1 0 0 1 0 0
1	0	0	0	0	0	0	1 0 0 0 0 0 0
0	0	1	0	0	0	0	0 0 1 0 0 0 0
0	0	1	1	0	0	0	0 0 1 1 0 0 0
0	1	0	0	1	0	0	0 1 0 0 1 0 0
0	1	0	1	1	0	0	0 1 0 1 1 0 0
0	1	0	0	0	0	0	0 1 0 0 0 0 0
0	0	0	1	1	0	0	0 0 0 1 1 0 0
1	0	0	0	1	0	0	1 0 0 0 1 0 0
0	0	0	0	0	0	1	0 0 0 0 0 0 1
0	0	0	0	1	1	0	0 0 0 0 1 1 0
0	0	0	1	0	1	0	0 0 0 1 0 1 0
0	0	0	0	0	1	0	0 0 0 0 0 1 0
1	0	1	0	1	0	0	1 0 1 0 1 0 0
0	0	1	0	1	0	0	0 0 1 0 1 0 0
0	0	1	0	1	1	0	0 0 1 0 1 1 0
1	1	0	0	1	1	0	1 1 0 0 1 1 0
1	0	1	0	0	0	0	1 0 1 0 0 0 0
0	0	0	1	0	0	1	0 0 0 1 0 0 1

FinalDatasetAll.csv							
Vulnerability Types							Category
Arbitrary_Memory_Access	Assertion_Failure	Block_Dependency	etherlock	integer overflow/underflow	reentrancy	TODAmount	Category
0	0	0	1	1	1	0	0 0 0 1 1 1 0
0	1	0	0	1	0	1	0 1 0 0 1 0 1
0	0	1	0	0	0	1	0 0 1 0 0 0 1
1	0	0	0	0	0	1	1 0 0 0 0 0 1
0	1	1	0	1	0	0	0 1 1 0 1 0 0
0	1	0	0	1	1	0	0 1 0 0 1 1 0
0	0	1	1	1	0	0	0 0 1 1 1 0 0
0	1	0	1	0	0	0	0 1 0 1 0 0 0

There are seven labels that correlate to a contract, and the value of each label is either 0 or 1. When the value is 0, it indicates that the contract does not have a particular vulnerability, and when the value is 1, it indicates that the contract does have a weakness of that sort. The vulnerability labels are different from one another. For instance, the label of the contract X is Lablex = [0 1 0 1 0 0 0], which indicates that the contract x has an Assertion_Failure and an Etherlock vulnerability.

We collected a total of 2194 verified smart contracts from the Etherscan.io website and manually added labels to all contracts based on the detection results given by Slither, Confuzzion, MAIAN, Conkas, and Securify static detection tools. The detection capacity of the model outlined in this thesis extends to identifying a total of seven distinct vulnerability varieties including: Arbitrary_Memory_Access, Assertion_Failure, Block_Dependency, etherlock, integer overflow/underflow, reentrancy, and TODAmount as shown in **Figure 24**.

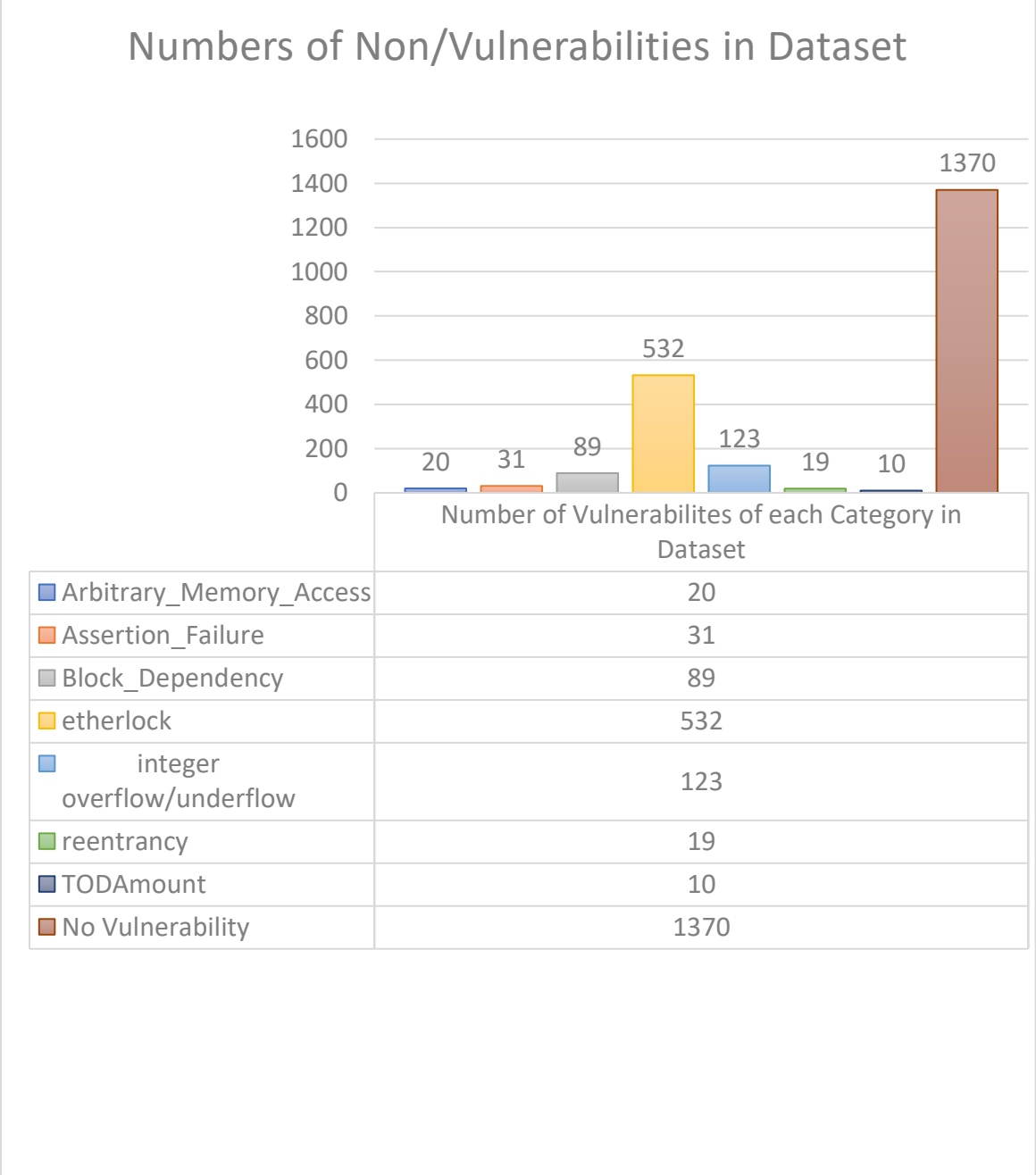

Figure 24: Numbers of Vulnerabilities in Dataset

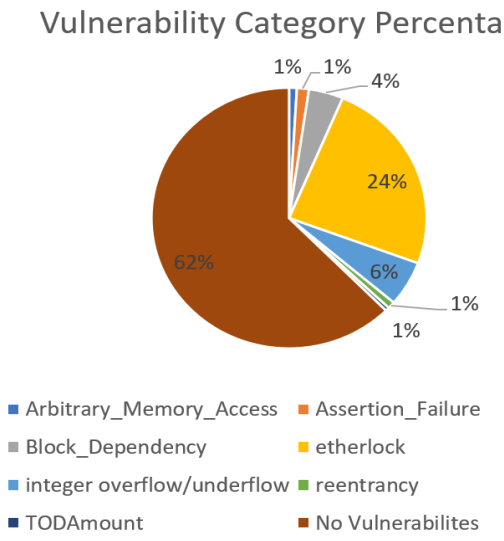

Figure 25: Vulnerability Percentage of Each Category in Dataset.

As an observation, we can see in **Figure 24** and **Figure 25** that 62% of our smart contract dataset is not vulnerable. The most detected vulnerability is “etherlock” with 24%, the second and third highest detected vulnerabilities categories are Integer overflow/underflow and Block_Dependency with 6% and 4% respectively. The rest of the attacks have almost a 1% detection ratio each.

3.1.5 INTERSECTION TECHNIQUE

During our analysis of the literature, we discovered that the accuracy of current vulnerability detection methods for Solidity contracts varied greatly, with some tools producing significant false positive rates and others failing to identify known issues [87] [88].

We used an intersection-based technique to increase the accuracy of our machine learning vulnerability detection model by only evaluating vulnerabilities reported by several tools. Through using this technique, we were able to decrease the risk of false positives and raise the overall accuracy of our model. Moreover, we were able to discover a broader variety of possible vulnerabilities in Solidity contracts.

This method also allowed us to learn about the most frequent types of vulnerabilities and their prevalence in real-world contracts, which helped us better understand the security concerns connected with these smart contracts.

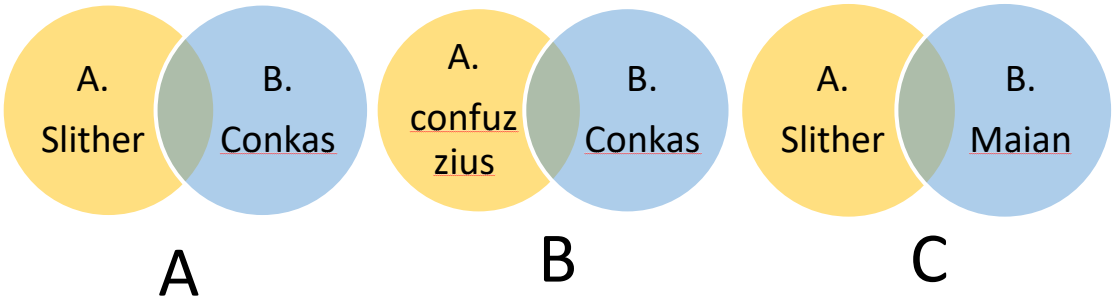

Figure 26: Tools Intersections

In **Figure 26** we can see the intersections between different smart contract static detection tools for identifying specific types of vulnerabilities. Specifically, we can see that we have taken the intersection between Slither and Maian to decrease false positives and negatives of the Etherlock vulnerability, and we have taken the intersection between Slither and Conkas to decrease false positives and negatives of the Reentrancy vulnerability. Additionally, we have taken the intersection between Confuzzius and Conkas to decrease false positives and negatives of the Integer Overflow/Underflow vulnerability.

These intersections represent the overlap between the results of different smart contract static detection tools for identifying vulnerabilities. By taking the intersection of multiple tools, we can reduce the number of false positives and negatives, which improves the accuracy and reliability of our vulnerability detection.

By using multiple smart contract static tools and taking the intersection of their results, we can improve the effectiveness of our vulnerability detection and reduce the risk of successful attacks. This approach is particularly useful for identifying complex and hard-to-detect vulnerabilities, which are increasingly common in today's threat landscape.

Since we found three different intersections between four tools, We have generated three sub-datasets as shown in **Table 7**, **Table 8**, and **Table 9** from our main dataset, namely integerconcat.csv, reentrancyconcat.csv, and etherlockconcat.csv. these three datasets were generated by taking the intersection of the results based on static tools. The main reason for this approach is to increase the accuracy and reduce false positive rates, which can improve the effectiveness of our research methodology. Moreover, this approach allows us

to identify if any specific vulnerabilities category resulting from static tools may require enhancement in terms of false positives.

Table 7: Integer Intersection Dataset Category.

IntegerConcat.csv	
integer overflow/underflow	Category
If Exist	1 0
Not Exist	0 0

Table 8: Reentrancy Intersection Dataset Category.

reentracncyconcat.csv	
Reentrancy	Category
If Exist	1 0
Not Exist	0 0

Table 9: Etherlock Dataset Category.

etherlockconcat.csv	
etherlockconcat	Category
If Exist	1 0
Not Exist	0 0

Chapter 4: RESULTS AND DISCUSSION

In this section, we will cover the various parameters used in the experimental methodology. Including the dataset used, parameter settings, and evaluation indicators. We will also present the experimental results and observations, providing a comprehensive and objective assessment of the research findings.

4.1 EXPERIMENT

The dataset consists of four columns, the first column is the smart contract address which the user can use to verify the existence of the smart contract on the etherscan.io website. The second column is the contract name, and the most two important columns in our experiment are the third and fourth which are opcode and the category of the vulnerabilities where these columns are used as machine learning inputs for detection.

Table 10 Dataset Columns Description.

Column Name	Column Description
ADDRESS	This column contains the smart contract address as per published in Etherscan.io. This column is not used as input or output in our experiment, it is only there for verification of the existence of smart contract on Etherscan.io
CONTRACTNAME	This column contains the contract name as shown on the Etherscan.io website. This column is not used as input or output, just to match the contract address with the name, and to know exactly what is this contract about, ex: if the contract name is "AMAZON" this will give an indication that the smart contract is about items related to amazon.
OPCODE	This column is used as input for the machine learning based detection process, and it contains the hexadecimal opcode value of the smart contract.
CATEGORY	This column presents the vulnerability categories for each smart contract.

4.1.1 DATASET AND PARAMETER SETTINGS

Our dataset was collected from the Ethereum official website, as it acquires validated smart contract codes to create a trustworthy experimental dataset. We added the multi label vectors to each smart contract manually, based on results conducted using static detection techniques.

Out of 2747 smart contracts that we have collected as CSV files and after the revision process, we found that there are 543 duplicated smart contracts and 10 contracts without an opcode. After we removed both duplicated and missing opcode smart contracts, 2194 remained in the dataset. **Table 11** shows the number of contracts in the final dataset that include each vulnerability.

Table 11: Number of smart contracts in Full Dataset.

Vulnerability Category	**Number of Vulnerability**
No Vulnerabilities	1370
Arbitrary_Memory_Acces	20
Assertion_Failure	31
Block_Dependency	89
Etherlock	532
Integer overflow/underflow	123
Reentrancy	19
TODAmount	10
Total Number	2194

The experiments were run on a powerful server to handle the load of the huge number of smart contracts. The specifications of the device are as follows:

- CPU:
    - Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz 2.29 GHz
    - 16 Cores
    - 32 Logical Processors
- Memory:
    - 128 GB
- OS:
    - Windows 10 Enterprise 64-bit
    - Ubuntu 22.04
- Software used:
    - Anaconda (Jupyter)
    - Python version: 3.9.12
    - Ubuntu Terminal
    - Tools cloned from GitHub.

We have used four different machine learning algorithms namely Decision Tree, Support Vector Machine, NN, and LSTM. The parameters setting for those algorithms were as follows, 80% of the data was used for training, while the remaining 20% was used for testing, based on a random distribution. More detailed settings for LSTM are as follows, the epochs and batch size were set to 50, and 63 respectively. The dataset is tested against each of the four algorithms. Moreover, the characteristics including accuracy, F1 score, precision, and recall were used to compare and evaluate them.

4.1.2 EVALUATION INDICATORS

Measures like accuracy, precision, recall, confusion matrix, and F-1 score are frequently used to gauge how effective machine learning is [112].

- **Accuracy** is the percentage of accurate predictions over all samples or possible predictions.

  Accuracy =TP + TN/TP + TN + FP + FN

- **Precision** is calculated as the ratio of true positives to both true and false positives.

  Precision =TP/TP + FP

- **Recall** score is a metric used to assess a model's performance by determining the percentage of true positives that were accurately detected. It is as follows:

  Recall=TP/TP + FN

- **The F-1 score** is the harmonic mean of precision and recall.

  F1=2*precision*recall/precision + recall

- **Confusion Matrix** is a table that demonstrates which values the model beliefs to be associated with various classes, allowing one to visualize the model's performance as shown in **Table 12**. With the rows representing the projected classes and the columns reflecting the actual classes, it has a size of N×N [112].

Multilabel classification was performed in the experiments of this thesis. The number of samples where both the predicted value and true value are negative is represented by TN (True Negative) in these experiments. The number of samples where the predicted value is negative and the true value is positive is represented by FN (False Negative). The predicted value, TP (True Positive), represents the number of samples where all true values are positive classes. The number of samples where predicted values are positive classes and the true values are negative classes is represented by FP (False Positive).

Table 12: Confusion Matrix

Actual \ **Predicated**	**One**	**Zero**
One	TP	FN
Zero	FP	TN

4.1.3 EXPERIMENTAL RESULTS AND OBSERVATION

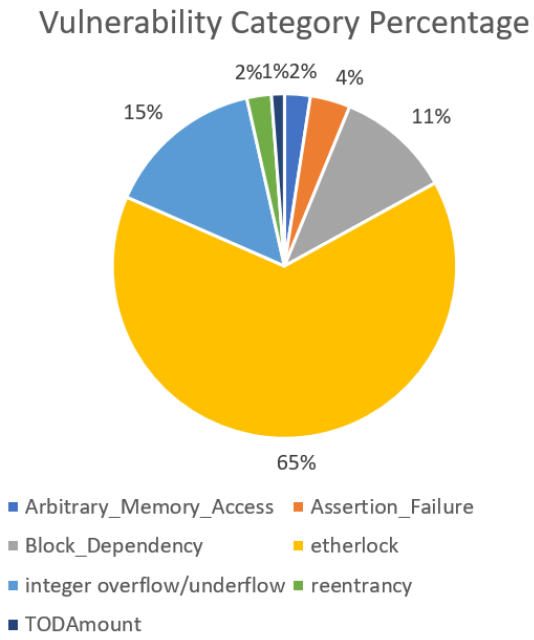

Figure 27: Vulnerability Category Percentage

After calculating the number of smart contracts that belongs to each vulnerability, we figured out that the most common vulnerability in our dataset is etherlock with a ratio of 65% of the total number of vulnerabilities. Furthermore, integer overflow/underflow and block dependency are 15% and 11% respectively. The rest of the vulnerabilities are less than or equal to 4% as shown in **Figure 27**.

Table 13: Precision, recall, and F1-score of Dataset which includes all vulnerabilities

Model	Indicator (%)	FullDataset	Reentrancy Dataset	Etherlock Dataset	Integer overflow/underflow Dataset
SVM	Precision	0.85	0.97	0.95	1.00
	Recall	0.98	0.95	0.76	1.00
	F1-score	0.91	0.96	0.84	1.00
DT	Precision	0.83	0.94	0.93	0.93
	Recall	0.87	0.85	0.74	0.75
	F1-score	0.85	0.89	0.82	0.83
NN	Precision	0.85	0.94	0.77	1.00
	Recall	0.90	0.85	0.71	1.00
	F1-score	0.87	0.89	0.74	1.00
LSTM	Precision	0.82	0.98	0.94	1.00
	Recall	0.92	0.96	0.76	1.00
	F1-score	0.87	0.97	0.84	1.00

Table 14: Accuracy and Time Cost

Model	Indicator	FullDataset	Reentrancy Dataset	Etherlock Dataset	Integer overflow/underflow Dataset
SVM	Accuracy	85.7	95.4	80.7	100.0
	Time Cost	15.54 S	0.03 S	0.21 S	0.15 S
DT	Accuracy	75.7	84.2	75.0	75.0
	Time Cost	8.44 S	0.11 S	0.08 S	0.04 S
NN	Accuracy	80.1	84.3	66.6	100.0
	Time Cost	5.26 S	0.10 S	0.15 S	0.04 S
LSTM	Accuracy	82.10	97.7	80.9	100.0
	Time Cost	240.78 S	14. 86 S	0.111 S	12.39 S

We noticed that using one static analysis tool to analyze a Solidity smart contract and then feeding the output of one tool into a machine learning algorithm means that we are tied to the efficiency of the static analysis tool. However, by feeding the output of multiple tools into a machine learning algorithm, we can potentially identify patterns and correlations in the results that may not be immediately apparent from an individual tool output. Consequently, more accurate results can be achieved. While each tool may have its own strengths and limitations, by using multiple tools and comparing their results, we can leverage their strengths and minimize their weaknesses. Also, this will reduce the number of false positives.

We have created a sub-dataset for Etherlock vulnerability as the accuracy results of this vulnerability in the static tools used are 69% for MAIAN specifically for Etherlock [40]. On the other hand, slither has 89% for all four vulnerabilities it detects. The authors of Slither tools did not mention the details for each vulnerability [87]. The intersection of results from these two tools results in having an accuracy of 80.7% in our machine learning detection technique. Having a sub-dataset helps researchers and companies who are targeting a specific vulnerability.

We had overfitting in our dataset specifically in the DT algorithm, and this is due to having a high number of specific records for some categories such as Etherlock and No vulnerability, compared with a few numbers of records for other categories such as Arbitrary_Memory_Access, Assertion_Failure, and Block_Dependency. We have solved overfitting for the reentrancy attack as we generated a sub-dataset that includes the intersection of two tools that can detect it. However, other attack categories are only detectable by only one tool which is Confuzzius. this limitation represents a challenge for solving the overfitting problem for these categories of attacks.

In **Table 13**, that both integer overflow and reentrancy sub datasets have high numbers in terms of recall, precision, and F1-score results. However, the etherlock dataset has low ratios of the evaluation indicators and this is due to false postives of static tools inherent deficiency. This can be targeted as future research to investigate the performance issues with Maian and Slither tools for etherlock vulnerability. On the other hand, we need to take into consideration that these tools jointly allows for higher ratio accuracy rates, i.e. 90% for other different vulnerability categories, using the LSTM algorithm.

The accuracy results of the four algorithms in the main dataset, as shown in **Table 14** are almost in average of 84%. This is due to the fact that the static tools used have a false

positive and false negative rate. This means they sometimes identified vulnerabilities that were not actual vulnerabilities, or missed vulnerabilities that were present.

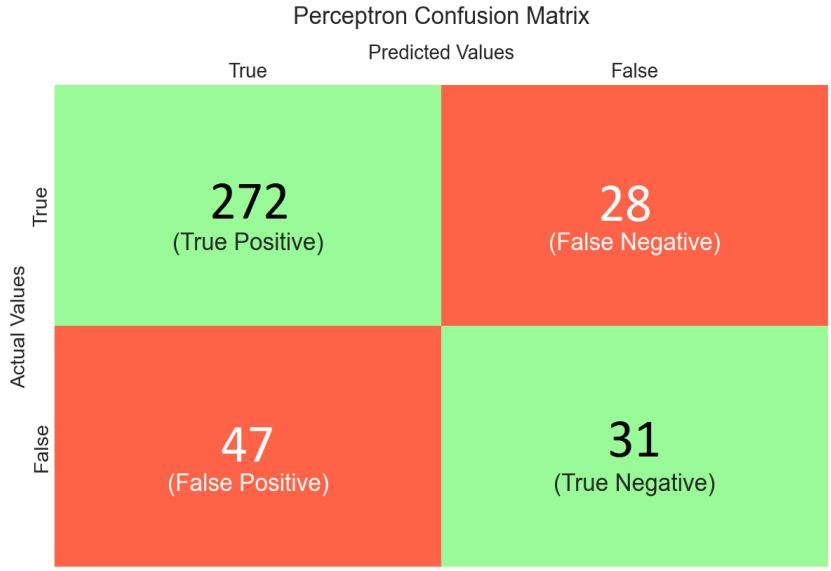

Figure 28: Perceptron Confusion Matrix

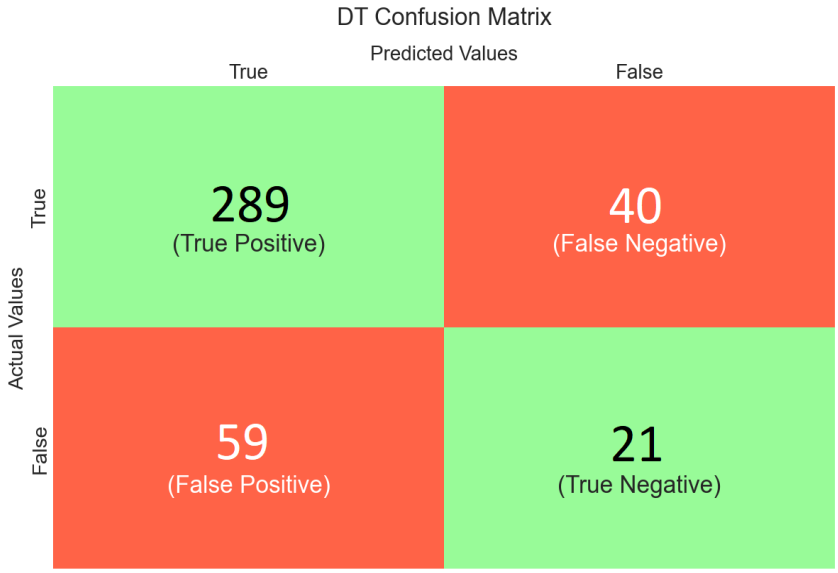

Figure 29: DT Confusion Matrix

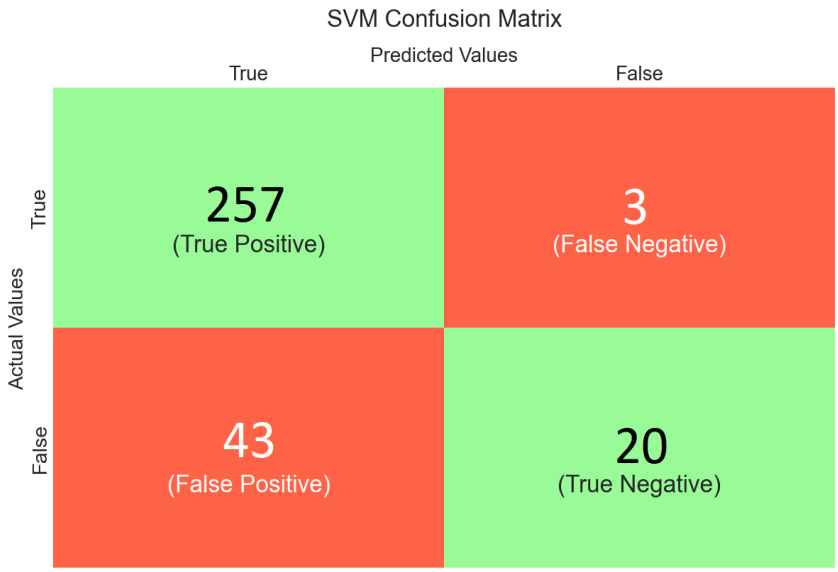

Figure 30: SVM Confusion Matrix

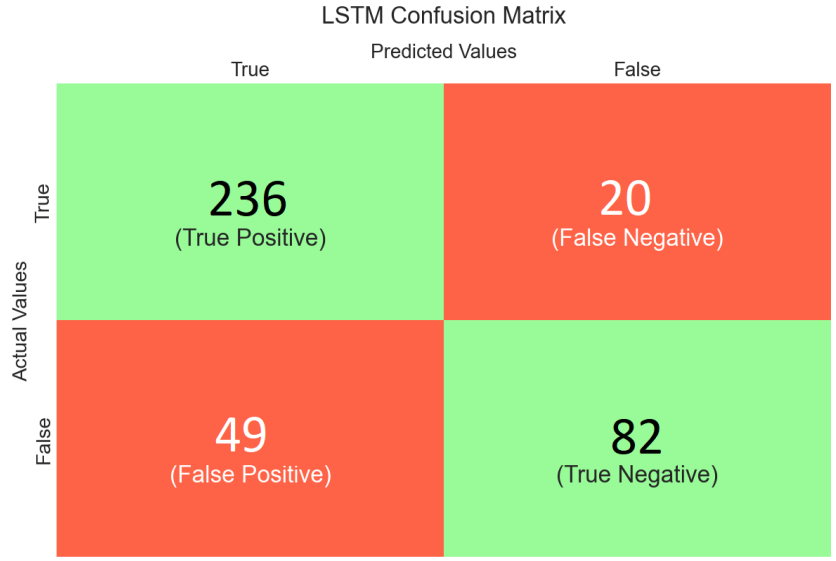

Figure 31: LSTM Confusion Matrix

In the confusion matrix, as shown in **Figure 28**, **Figure 29**, **Figure 30**, and **Figure 31**, Show the Confusion Matrix, LSTM has fewer False Positive than other algorithms, which indicates a good prediction methodology although it has a bit more False-Negative than others. SVM also achieved the best rate in terms of False Negative as it achieved a 0 rate. However, in terms of False Positive it has a higher rate than LSTM. On the other hand, NN and Decision Tree have almost the same results.

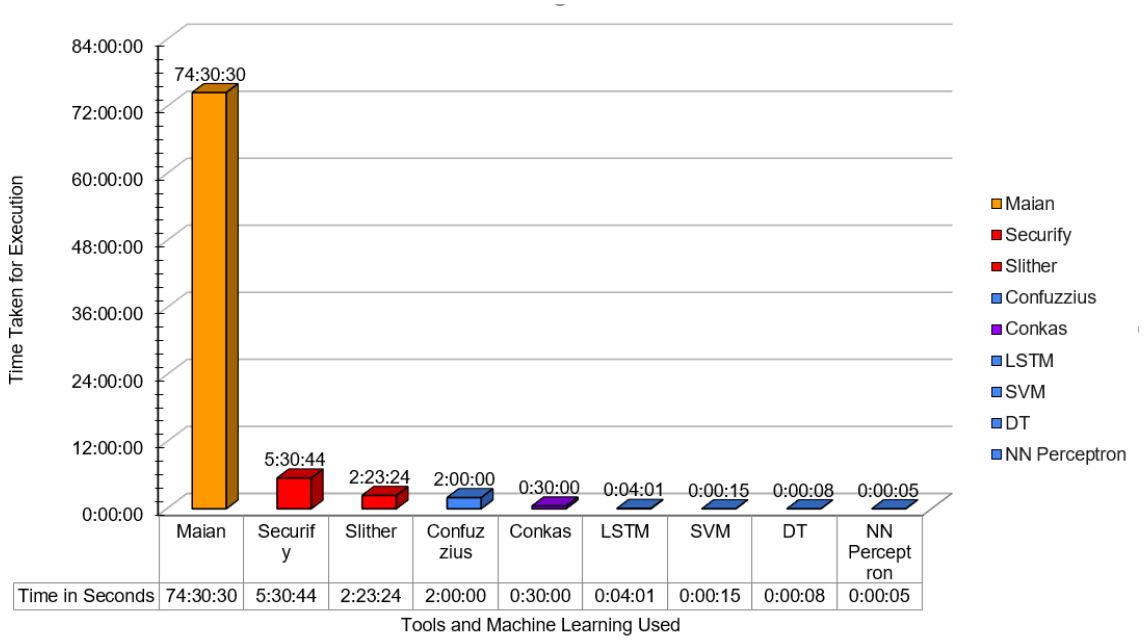

Figure 32: Comparing time taken by each Tool.

Table 15: Total Time taken by each methodology

Tool Name	Time Taken Per 500 Contracts	Total Time Taken for full dataset (2194) SC
Maian	**74:30:30 * 4.338**	**322:37:16**
Securify	**5:30:44 *4.338**	**23:52:5**
Slither	**2:23:24 * 4.338**	**10:20:55**
Confuzziuz	**2:00:00 * 4.338**	**08:39:36**
Conkas	**0:30:00 * 4.338**	**02:09:54**
LSTM	**-**	**00:04:01**
SVM	**-**	**00:00:15**
DT	**-**	**00:00:08**
NN Perceptron	**-**	**00:00:05**

Before running the smart contract on static tools, we have divided the dataset into almost four sub-dataset, so we do have in each dataset around 500 smart contracts. This allows us to overcome the challenge that we had when we ran the whole big dataset without reducing its size via division. The problem is related to the fact that the virtual machine was not able to handle the dataset as a whole and the machine was always forced to crash. Therefore the results in **Figure 31** are the time taken for static tools to get results of around 500 smart

contracts. However, **Figure 31** also includes our machine learning algorithms where we run all smart contracts “2194”.

As shown in **Figure 31**, all machine learning Time Execution “Time Cost” Compared to static tools have a much lower time cost. This is because they can analyze large amounts of data in a relatively short amount of time and make predictions based on given data. Static tools, on the other hand, are typically limited to processing a fixed set of rules or criteria, and they may not be able to adapt to changing data patterns.

Another reason why machine learning is faster than static tools is its ability to handle complex data sets. They can identify patterns in the data that may be difficult or impossible for static tools to detect. This makes them particularly useful for applications such as image recognition, speech recognition, and natural language processing.

Machine learning algorithms have transformed the way we analyze data and have provided faster and more accurate results compared to traditional static tools. As data sets continue to grow in size and complexity, the use of machine learning algorithms will become increasingly important for businesses and organizations that want to gain insights from their data in a timely and cost effective manner.

Chapter 5: CONCLUSION AND FUTURE WORK

Smart contracts are self executing contracts that are programmed to automatically execute when certain predefined conditions are met. They are integral to the functioning of blockchain systems. Since blockchain is decentralized and immutable, smart contracts cannot be changed or modified after deployment. Therefore, there is a need to check the smart contract by analyzing it specifically at the opcode level, where the developers can understand the functions and logic of the code and identify potential vulnerabilities and implement measures to prevent them.

A comprehensive survey of attack detection techniques used in smart contracts is provided, which includes static analysis, dynamic analysis, and hybrid approaches. Additionally, we analyzed the benefits and drawbacks of each method and conducted a comparative evaluation of the current instruments employed for various smart contract analysis methods. In addition, our approach for detecting attacks on smart contracts is based on machine learning. A tool was developed to collect data from etherscan.io, which was previously unavailable. Static detection tools were used to test the data after collecting the dataset. The machine learning algorithms were fed with manually multi-labeled results from these tools. To enhance the precision of the dataset, this process serves its purpose.

The use of machine learning algorithms such as SVM, LSTM, DT, and NN has shown significant results in detecting vulnerabilities in smart contracts in our detecting framework. These algorithms have been able to accurately identify vulnerabilities in smart contracts, resulting in high recall, precision, and F1-score results. However, it is important to note that false positives can still occur, which can lead to a decrease in the recall, precision, and F1-score results as the in Etherlock sub dataset.

Results shown for four ML models, namely Decision Tree, Perceptron, Support Vector Machine (SVM), and Long Short-Term Memory (LSTM) are used for this research based on final datasets and sub dataset and the best accuracy results for full dataset 85.7% using SVM, Reentrancy dataset 97.7% using LSTM, Etherlock dataset 80.9% using LSTM, integer overflow/underflow dataset 100% using SVM, Perceptron, and LSTM.

In terms of time cost, SVM, DT, and NN took less than 15 seconds to get the results of the main dataset,however, LSTM requires longer training times due to its complexity as it took 4 minutes. Thus, the choice of algorithm should consider both accuracy and time cost. LSTM was the highest algorithm in terms of accuracy but the lowest in terms of Time cost.

5.1 FUTURE WORK.

The following should be the focus of future research:

- Increasing the number of records in the generated datasets.
- Designing and implementing more efficient vulnerability detection tools for better accuracy results.
- Adding more vulnerability categories.

References

[1] A. Urquhart, "The inefficiency of Bitcoin," *Econ Lett*, vol. 148, pp. 80–82, Nov. 2016, doi: 10.1016/j.econlet.2016.09.019.

[2] S. Makridakis and K. Christodoulou, "Blockchain: Current challenges and future prospects/applications," *Future Internet*, vol. 11, no. 12. MDPI AG, Dec. 01, 2019. doi: 10.3390/FI11120258.

[3] A. Hackethal, T. Hanspal, D. M. Lammer, and K. Rink, "The Characteristics and Portfolio Behavior of Bitcoin Investors: Evidence from Indirect Cryptocurrency Investments," *Rev Financ*, vol. 26, no. 4, pp. 855–898, Jul. 2022, doi: 10.1093/rof/rfab034.

[4] J. Carrick, "Bitcoin as a Complement to Emerging Market Currencies," *Emerging Markets Finance and Trade*, vol. 52, no. 10, pp. 2321–2334, Oct. 2016, doi: 10.1080/1540496X.2016.1193002.

[5] D. Jain, "Emerging Blockchain Technology in Commercial Enterprise to Ensure Electronic Revolution: Challenges and Improvement." [Online]. Available: www.pbr.co.in

[6] E. A. Boakye, H. Zhao, and B. N. K. Ahia, "Emerging research on blockchain technology in finance; a conveyed evidence of bibliometric-based evaluations," *Journal of High Technology Management Research*, vol. 33, no. 2, Nov. 2022, doi: 10.1016/j.hitech.2022.100437.

[7] M. Qatawneh, W. Almobaideen, and O. AbuAlghanam, "Challenges of Blockchain Technology in Context Internet of Things: A Survey," *Int J Comput Appl*, vol. 175, no. 16, pp. 13–20, Sep. 2020, doi: 10.5120/ijca2020920660.

[8] B. Alhasan, M. Qatawneh, and W. Almobaideen, "Blockchain Technology for Preventing Counterfeit in Health Insurance," in *2021 International Conference on Information Technology, ICIT 2021 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., Jul. 2021, pp. 935–941. doi: 10.1109/ICIT52682.2021.9491664.

[9] A. Benabdallah, A. Audras, L. Coudert, N. El Madhoun, and M. Badra, "Analysis of Blockchain Solutions for E-Voting: A Systematic Literature Review," *IEEE Access*, vol. 10, pp. 70746–70759, 2022, doi: 10.1109/ACCESS.2022.3187688.

[10] P. Sharma, R. Jindal, and M. D. Borah, "A review of smart contract-based platforms, applications, and challenges," *Cluster Comput*, vol. 26, no. 1, pp. 395–421, Feb. 2023, doi: 10.1007/s10586-021-03491-1.

[11] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F. Y. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Trans Syst Man Cybern Syst*, vol. 49, no. 11, pp. 2266–2277, Nov. 2019, doi: 10.1109/TSMC.2019.2895123.

[12] S. M. Skh Saad and R. Z. Raja Mohd Radzi, "Comparative Review of the Blockchain Consensus Algorithm Between Proof of Stake (POS) and Delegated Proof of Stake (DPOS)," *International Journal of Innovative Computing*, vol. 10, no. 2, Nov. 2020, doi: 10.11113/ijic.v10n2.272.

[13] A. K. Yadav, K. Singh, A. H. Amin, L. Almutairi, T. R. Alsenani, and A. Ahmadian, "A comparative study on consensus mechanism with security threats and future scopes: Blockchain," *Comput Commun*, vol. 201, pp. 102–115, Mar. 2023, doi: 10.1016/j.comcom.2023.01.018.

[14] M. S. Hossan, M. L. Khatun, S. Rahman, S. Reno, and M. Ahmed, "Securing Ride-Sharing Service Using IPFS and Hyperledger Based on Private Blockchain," in *24th International Conference on Computer and Information Technology, ICCIT 2021*, Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/ICCIT54785.2021.9689814.

[15] R. Belen-Saglam, E. Altuncu, Y. Lu, and S. Li, "A systematic literature review of the tension between the GDPR and public blockchain systems," *Blockchain: Research and Applications*, p. 100129, Jan. 2023, doi: 10.1016/j.bcra.2023.100129.

[16] S. Odeh, A. Samara, R. Rizqallah, and L. Shaheen, "Digital Identity Using Hyperledger Fabric as a Private Blockchain-Based System," in *Lecture Notes in Networks and Systems*, Springer Science and Business Media Deutschland GmbH, 2023, pp. 153–161. doi: 10.1007/978-3-031-21229-1_15.

[17] K. Qian, Y. Liu, X. He, M. Du, S. Zhang, and K. Wang, "HPCchain: A Consortium Blockchain System based on CPU-FPGA Hybrid PUF for Industrial Internet of Things," *IEEE Trans Industr Inform*, 2023, doi: 10.1109/TII.2023.3244339.

[18] Institute of Electrical and Electronics Engineers, *Probing artificial intelligence techniques and research in IoT (PATRIOT) 5th International Conference on Science, Technology, Engineering & Mathematics : IEEE ICONSTEM '19 : 15th*

March 2019, venue: Placement Seminar Hall, Jeppiaar Engineering College, Chennai.

[19] M. Yano, C. Dai, K. Masuda, and Y. Kishimoto, "Economics, Law, and Institutions in Asia Pacific Blockchain and Crypt Currency Building a High Quality Marketplace for Crypt Data." [Online]. Available: http://www.springer.com/series/13451

[20] V. Dhillon, D. Metcalf, and M. Hooper, "Unpacking Ethereum," in *Blockchain Enabled Applications*, Apress, 2021, pp. 37–72. doi: 10.1007/978-1-4842-6534-5_4.

[21] R. Dennis and J. P. Disso, "An analysis into the scalability of bitcoin and ethereum," in *Advances in Intelligent Systems and Computing*, Springer Verlag, 2019, pp. 619–627. doi: 10.1007/978-981-13-1165-9_57.

[22] Institute of Electrical and Electronics Engineers, *2020 IEEE International Conference on Communications : proceedings : Dublin, Ireland, 7-11 June 2020.*

[23] C. Saraf and S. Sabadra, "Blockchain platforms: A compendium," in *2018 IEEE International Conference on Innovative Research and Development, ICIRD 2018*, Institute of Electrical and Electronics Engineers Inc., Jun. 2018, pp. 1–6. doi: 10.1109/ICIRD.2018.8376323.

[24] G. A. F. Rebello, G. F. Camilo, L. C. B. Guimaraes, L. A. C. De Souza, and O. C. M. B. Duarte, "Security and Performance Analysis of Quorum-based Blockchain Consensus Protocols," in *2022 6th Cyber Security in Networking Conference, CSNet 2022*, Institute of Electrical and Electronics Engineers Inc., 2022. doi: 10.1109/CSNet56116.2022.9955597.

[25] IEEE Computer Society, Institute of Electrical and Electronics Engineers., P. IEEE/ACM International Conference on Cyber, IEEE/ACM International Conference on Green Computing and Communications (16th : 2020 : Online), IOT (Conference) (13th : 2020 : Online), and IEEE International Conference on Smart Data (6th : 2020 : Online), *IEEE Congress on Cybermatics ; 2020 IEEE International Conferences on Internet of Things (iThings) ; IEEE Green Computing and Communications (GreenCom) ; IEEE Cyber, Physical and Social Computing (CPSCom) ; IEEE Smart Data (SmartData) : Cybermatics 2020, iThings 2020, GreenCom 2020, CPSCom 2020, SmartData 2020 : proceedings : Rhodes Island, Greece, 2-6 November 2020.*

[26] M. Suvitha and R. Subha, "A Survey on Smart Contract Platforms and Features," in *2021 7th International Conference on Advanced Computing and Communication Systems, ICACCS 2021*, Institute of Electrical and Electronics

Engineers Inc., Mar. 2021, pp. 1536–1539. doi: 10.1109/ICACCS51430.2021.9441970.

[27] T. M. Hewa, Y. Hu, M. Liyanage, S. S. Kanhare, and M. Ylianttila, "Survey on Blockchain-Based Smart Contracts: Technical Aspects and Future Research," *IEEE Access*, vol. 9. Institute of Electrical and Electronics Engineers Inc., pp. 87643–87662, 2021. doi: 10.1109/ACCESS.2021.3068178.

[28] L. M. Palma, M. A. G. Vigil, F. L. Pereira, and J. E. Martina, "Blockchain and smart contracts for higher education registry in Brazil," in *International Journal of Network Management*, John Wiley and Sons Ltd, May 2019. doi: 10.1002/nem.2061.

[29] V. Y. Kemmoe, W. Stone, J. Kim, D. Kim, and J. Son, "Recent Advances in Smart Contracts: A Technical Overview and State of the Art," *IEEE Access*, vol. 8, pp. 117782–117801, 2020, doi: 10.1109/ACCESS.2020.3005020.

[30] I. H. Sarker, "Machine Learning: Algorithms, Real-World Applications and Research Directions," *SN Computer Science*, vol. 2, no. 3. Springer, May 01, 2021. doi: 10.1007/s42979-021-00592-x.

[31] L. Alzubaidi *et al.*, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *J Big Data*, vol. 8, no. 1, Dec. 2021, doi: 10.1186/s40537-021-00444-8.

[32] R. Montasari, F. Carroll, S. Macdonald, H. Jahankhani, A. Hosseinian-Far, and A. Daneshkhah, "Application of Artificial Intelligence and Machine Learning in Producing Actionable Cyber Threat Intelligence," in *Advanced Sciences and Technologies for Security Applications*, Springer, 2021, pp. 47–64. doi: 10.1007/978-3-030-60425-7_3.

[33] E. E. Abdallah, W. Eleisah, and A. F. Otoom, "Intrusion Detection Systems using Supervised Machine Learning Techniques: A survey," in *Procedia Computer Science*, Elsevier B.V., 2022, pp. 205–212. doi: 10.1016/j.procs.2022.03.029.

[34] E. Tufan, C. Tezcan, and C. Acartürk, "Anomaly-based intrusion detection by machine learning: A case study on probing attacks to an institutional network," *IEEE Access*, vol. 9, pp. 50078–50092, 2021, doi: 10.1109/ACCESS.2021.3068961.

[35] Institute of Electrical and Electronics Engineers, *2020 IEEE Symposium on Computers and Communications (ISCC)*.

[36] B. Hu *et al.*, "A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems," *Patterns*, vol. 2, no. 2. Cell Press, Feb. 12, 2021. doi: 10.1016/j.patter.2020.100179.

[37] R. F. Ibrahim, Q. Abu Al-Haija, and A. Ahmad, "DDoS Attack Prevention for Internet of Thing Devices Using Ethereum Blockchain Technology," *Sensors*, vol. 22, no. 18, Sep. 2022, doi: 10.3390/s22186806.

[38] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7. Institute of Electrical and Electronics Engineers Inc., pp. 150184–150202, 2019. doi: 10.1109/ACCESS.2019.2946988.

[39] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," Jun. 2018, [Online]. Available: http://arxiv.org/abs/1806.01143

[40] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," Feb. 2018, [Online]. Available: http://arxiv.org/abs/1802.06038

[41] W. Zhang, V. Ganesh, S. Banescu, L. Pasos, and S. Stewart, "MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract."

[42] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, Inc, Jul. 2020, pp. 415–427. doi: 10.1145/3395363.3397385.

[43] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the ACM Conference on Computer and Communications Security*, Association for Computing Machinery, Nov. 2019, pp. 531–548. doi: 10.1145/3319535.3363230.

[44] N. F. Samreen and M. H. Alalfi, "SmartScan: An approach to detect Denial of Service Vulnerability in Ethereum Smart Contracts," in *Proceedings - 2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2021*, Institute of Electrical and Electronics Engineers Inc., May 2021, pp. 17–26. doi: 10.1109/WETSEB52558.2021.00010.

[45] M. Maffei and M. Ryan, Eds., *Principles of Security and Trust*, vol. 10204. in Lecture Notes in Computer Science, vol. 10204. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. doi: 10.1007/978-3-662-54455-6.

[46] L. Duan, Y. Sun, K. Zhang, and Y. Ding, "Multiple-Layer Security Threats on the Ethereum Blockchain and Their Countermeasures," *Security and Communication Networks*, vol. 2022, 2022, doi: 10.1155/2022/5307697.

[47] T. Chen *et al.*, "SODA: A Generic Online Detection Framework for Smart Contracts," Internet Society, Feb. 2020. doi: 10.14722/ndss.2020.24449.

[48] "https://swcregistry.io/docs/SWC-124" , available online last accessed 5/5/2023.

[49] Y. Wang *et al.*, "Formal Specification and Verification of Smart Contracts for Azure Blockchain," Dec. 2018, [Online]. Available: http://arxiv.org/abs/1812.08829

[50] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey," Aug. 2019, [Online]. Available: http://arxiv.org/abs/1908.08605

[51] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," Feb. 2018, [Online]. Available: http://arxiv.org/abs/1802.06038

[52] "https://redfoxsec.com/blog/integer-overflow-in-smart-contract/" , available online last accessed 5/5/2023.

[53] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts," in *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, Institute of Electrical and Electronics Engineers Inc., Sep. 2020, pp. 1029–1040. doi: 10.1145/3324884.3416553.

[54] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the ACM Conference on Computer and Communications Security*, Association for Computing Machinery, Oct. 2018, pp. 67–82. doi: 10.1145/3243734.3243780.

[55] "https://swcregistry.io/docs/SWC-124#description" , available online last accessed 5/5/2023.

[56] “https://swcregistry.io/docs/SWC-110#description” , available online last accessed 5/5/2023.

[57] “https://swcregistry.io/docs/SWC-116#description” , available online last accessed 5/5/2023.

[58] “https://swcregistry.io/docs/SWC-132#description” , available online last accessed 5/5/2023.

[59] “https://swcregistry.io/docs/SWC-101#description” , available online last accessed 5/5/2023.

[60] “https://swcregistry.io/docs/SWC-107#description” , available online last accessed 5/5/2023.

[61] “https://swcregistry.io/docs/SWC-114#description” , available online last accessed 5/5/2023.

[62] “https://swcregistry.io/docs/SWC-124#remediation” , available online last accessed 5/5/2023.

[63] “https://swcregistry.io/docs/SWC-110#remediation” , available online last accessed 5/5/2023.

[64] “https://swcregistry.io/docs/SWC-116#remediation” , available online last accessed 5/5/2023.

[65] “https://swcregistry.io/docs/SWC-132#remediation” , available online last accessed 5/5/2023.

[66] “https://swcregistry.io/docs/SWC-101#remediation” , available online last accessed 5/5/2023.

[67] “https://swcregistry.io/docs/SWC-107#remediation” , available online last accessed 5/5/2023.

[68] “https://swcregistry.io/docs/SWC-114#remediation” , available online last accessed 5/5/2023.

[69] “https://www.avax.network/”, available online last accessed 5/5/2023.

[70] “https://moonbeam.network/”, available online last accessed 5/5/2023.

[71] “https://polygon.technology/”, available online last accessed 5/5/2023.

[72] “https://remix.ethereum.org/”, available online last accessed 5/5/2023.

[73] “https://blog.logrocket.com/smart-contract-programming-languages/”.

[74] S. Bistarelli, G. Mazzante, M. Micheletti, L. Mostarda, D. Sestili, and F. Tiezzi, "Ethereum smart contracts: Analysis and statistics of their source code and opcodes," *Internet of Things (Netherlands)*, vol. 11, Sep. 2020, doi: 10.1016/j.iot.2020.100198.

[75] M. Suiche, "Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode," 2017.

[76] "https://ethereum.org/en/developers/docs/evm/opcodes/" , available online last accessed 5/5/2023.

[77] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, Inc, Jul. 2020, pp. 415–427. doi: 10.1145/3395363.3397385.

[78] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the ACM Conference on Computer and Communications Security*, Association for Computing Machinery, Nov. 2019, pp. 531–548. doi: 10.1145/3319535.3363230.

[79] N. F. Samreen and M. H. Alalfi, "SmartScan: An approach to detect Denial of Service Vulnerability in Ethereum Smart Contracts," in *Proceedings - 2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2021*, Institute of Electrical and Electronics Engineers Inc., May 2021, pp. 17–26. doi: 10.1109/WETSEB52558.2021.00010.

[80] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models," *IEEE Access*, vol. 8, pp. 19685–19695, 2020, doi: 10.1109/ACCESS.2020.2969429.

[81] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM Conference on Computer and Communications Security*, Association for Computing Machinery, Oct. 2016, pp. 254–269. doi: 10.1145/2976749.2978309.

[82] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," Jun. 2018, [Online]. Available: http://arxiv.org/abs/1806.01143

[83] “https://github.com/ConsenSys/mythril”, available online last accessed 5/5/2023.

[84] J. Krupp and C. Rossow, *Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX. teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts*. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[85] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale,” Feb. 2018, [Online]. Available: http://arxiv.org/abs/1802.06038

[86] B. Jiang, Y. Liu, and W. K. Chan, “ContractFuzzer: Fuzzing smart contracts for vulnerability detection,” in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Association for Computing Machinery, Inc, Sep. 2018, pp. 259–269. doi: 10.1145/3238147.3238177.

[87] J. Feist, G. Grieco, and A. Groce, “Slither: A Static Analysis Framework For Smart Contracts,” Aug. 2019, doi: 10.1109/WETSEB.2019.00008.

[88] N. Veloso and I. Superior Técnico, “Conkas: A Modular and Static Analysis Tool for Ethereum Bytecode,” 2021. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-con

[89] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, “ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts,” in *Proceedings - 2021 IEEE European Symposium on Security and Privacy, Euro S and P 2021*, Institute of Electrical and Electronics Engineers Inc., Sep. 2021, pp. 103–119. doi: 10.1109/EuroSP51992.2021.00018.

[90] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, “Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models,” *IEEE Access*, vol. 8, pp. 19685–19695, 2020, doi: 10.1109/ACCESS.2020.2969429.

[91] “https://github.com/gongbell/ContractFuzzer ”, available online last access 5/5/2023.

[92] “https://github.com/crytic/slither”, available online last accessed 5/5/2023.

[93] “https://github.com/nveloso/conkas”, available online last accessed 5/5/2023.

[94] “https://github.com/christoftorres/ConFuzzius”, available online last accessed 5/5/2023.

[95] “https://github.com/ivicanikolicsg/MAIAN”, available online last accessed 5/5/2023.

[96] “https://github.com/nescio007/teether”, available online last accessed 5/5/2023.

[97] “https://github.com/ConsenSys/mythril”, available online last accessed 5/5/2023.

[98] “https://github.com/enzymefinance/oyente”, available online last accessed 5/5/2023.

[99] “https://github.com/eth-sri/securify2”, available online last accessed 5/5/2023.

[100] J. Kang, S. Jang, S. Li, Y. S. Jeong, and Y. Sung, “Long short-term memory-based Malware classification method for information security,” *Computers and Electrical Engineering*, vol. 77, pp. 366–375, Jul. 2019, doi: 10.1016/j.compeleceng.2019.06.014.

[101] H. Fujiyoshi, T. Hirakawa, and T. Yamashita, “Deep learning-based image recognition for autonomous driving,” *IATSS Research*, vol. 43, no. 4. Elsevier B.V., pp. 244–252, Dec. 01, 2019. doi: 10.1016/j.iatssr.2019.11.008.

[102] Y. Liu, F. R. Yu, X. Li, H. Ji, and V. C. M. Leung, “Blockchain and Machine Learning for Communications and Networking Systems,” *IEEE Communications Surveys and Tutorials*, vol. 22, no. 2, pp. 1392–1431, Apr. 2020, doi: 10.1109/COMST.2020.2975911.

[103] A. V Joshi, “Machine Learning and Artii cial Intelligence.”

[104] F. E. B. Otero, A. A. Freitas, and C. G. Johnson, “Inducing decision trees with an ant colony optimization algorithm,” *Applied Soft Computing Journal*, vol. 12, no. 11, pp. 3615–3626, Nov. 2012, doi: 10.1016/j.asoc.2012.05.028.

[105] C. AVCI, M. BUDAK, N. YAĞMUR, and F. BALÇIK, “Comparison Between Random Forest and Support Vector Machine Algorithms for LULC Classification,” *International Journal of Engineering and Geosciences*, Nov. 2021, doi: 10.26833/ijeg.987605.

[106] A. Sherstinsky, “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network,” *Physica D*, vol. 404, Mar. 2020, doi: 10.1016/j.physd.2019.132306.

[107] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust Intelligent Malware Detection Using Deep Learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019, doi: 10.1109/ACCESS.2019.2906934.

[108] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A Survey of Android Malware Detection with Deep Neural Models," *ACM Computing Surveys*, vol. 53, no. 6. Association for Computing Machinery, Feb. 01, 2021. doi: 10.1145/3417978.

[109] X. Yang, D. Yang, and Y. Li, "A Hybrid Attention Network for Malware Detection Based on Multi-Feature Aligned and Fusion," *Electronics (Switzerland)*, vol. 12, no. 3, Feb. 2023, doi: 10.3390/electronics12030713.

[110] A. I. Elkhawas and N. Abdelbaki, "Malware Detection using Opcode Trigram Sequence with SVM; Malware Detection using Opcode Trigram Sequence with SVM," 2018.

[111] M. Fazlali, P. Khodamoradi, F. Mardukhi, M. Nosrati, and M. M. Dehshibi, "Metamorphic malware detection using opcode frequency rate and decision tree," *International Journal of Information Security and Privacy*, vol. 10, no. 3, pp. 67–86, Jul. 2016, doi: 10.4018/IJISP.2016070105.

[112] "Performance Measures for Machine Learning. https://www.javatpoint.com/performance-metrics-in-machine-learning"

Appendix A

Pseudo Code below describes how we collect Solidity Code.

```
# Start Chrome
SET driver TO webdriver.Chrome()

SET count TO 0   #To check the progress of retriveing smartcontract.
with open("contractaddresses.txt") as readContractAddress: #read smart contract addresses from contractaddresses.txt

   WHILE True:
      SET line TO readContractAddress.readline()
      IF not line:
         break
      count += 1
      OUTPUT("The Number of Solidity are "+ str(count) + " "+line)
      lline=line.strip("\n") # to remove the \n from the address as everytime it read the address it shows after it \n.



      # Open the webpage
      driver.get("https://etherscan.io/address/"+line+"#code")
      time.sleep(2) # 2 seconds to load the webpage
      SET html TO driver.page_source

      #write solidity address IF smartcontract have multi solidity file
      IF "File 1 of" IN html:
         OUTPUT("The address Contains Mutiple Sol files")
         with open("MultiSolidity.txt", 'a') as un:
            un.write(line)

      ELSE:
         SET buttton TO driver.find_element(By.ID, "panel-sourcecode")  ##To view all code by pressing on Full Screen
```

```
        buttton.click()
        SET FindSolidityText TO driver.find_element(By.ID, "editor") #To find the text area of the code
        SET code TO FindSolidityText.text #To get the text code

        #save solidity output IN .sol format
        soloutput="C:\\Users\\user\\Desktop\\solidity\\solidity"+str(lline)+".sol"
        with open(soloutput, 'w', encoding='utf-8') as writesoliditycode:
            writesoliditycode.write(code)
```

Pseudo Code below describes how we collect Bytecode and Opcode.

```
# Start Chrome

SET driver TO webdriver.Chrome()
SET count TO 0   #To check the progress of retriveing smartcontract.
countexcel= 0   #To determine the cell of CSV file.

with open("contractaddresses.txt") as readContractAddress: #read smart contract addresses from contractaddresses.txt
   WHILE True:
      count += 1
      countexcel += 1
      SET line TO readContractAddress.readline()

      IF not line:
         break

      lline=line.strip("\n") # to remove the \n from the address as everytime it read the address it shows after it \n.

      # Open the webpage
      driver.get("https://etherscan.io/address/"+line+"#code")
      time.sleep(15) # 15 seconds to load the webpage
```

```
SET elembytecode TO driver.find_element(By.ID, "verifiedbytecode2")  # To find the verifiedbytecode2 element
SET textbytecode TO elembytecode.text #to extract the text (bytecode) from the verifiedbytecode2 #note: text is a method IN driver


OUTPUT("Contract No. "+ str(count) +": Retrieving SmartContract " + lline)  # to OUTPUT contract number and each address.


#save bytecode output IN .bytecode format

bytecodeoutput="C:\\Users\\user\\Desktop\\bytecode\\bytecode"+str(lline)+".bytecode"
with open(bytecodeoutput, 'w') as writebytecode:
   writebytecode.write(textbytecode)

"""
#To Create list of bytecode files names.
string_to_check="bytecode"+str(lline)+".bytecode"
with open("name.txt", "r") as file:
   SET lines TO file.readlines()
   IF string_to_check not IN lines:
      with open("name.txt", "a") as file:
         file.write(string_to_check + "\n")
"""

# Wait FOR the opcodes to be loaded
time.sleep(5)

# Click the "Switch To Opcodes View" button
SET button TO driver.find_element(By.ID, "btnConvert3")
button.click()
```

```
# Retrieve the verifiedbytecode2 element
SET elem TO driver.find_element(By.ID, "verifiedbytecode2")
SET textopcodestring TO elem.text #to extract the text (opcode) from the verifiedbytecode2 #note: text is a method IN driver

#save opcodestring output IN .txt format

opcodeoutputstring="C:\\Users\\user\\Desktop\\opcodestring\\opcodestring"+str(lline)+".txt"
with open(opcodeoutputstring, 'w') as writeOpcodeOutputString:
   writeOpcodeOutputString.write(textopcodestring)
```

Pseudo Code below describes how to convert Opcode to Opcode Hexadecimal Values.

```
    #change opcodestring fromat into opcodehexadecimal format
with open(opcodeoutputstring, "r") as ReadOpcodeOutputString:
   SET wordlist TO [r.split()[0] FOR r IN ReadOpcodeOutputString]

   # define list
   SET i TO 0
   WHILE i < len(wordlist):
      IF wordlist[i] EQUALS 'STOP':
         SET wordlist[i] TO '00'
      IF wordlist[i] EQUALS 'ADD':
         SET wordlist[i] TO '01'
      IF wordlist[i] EQUALS 'MUL':
         SET wordlist[i] TO '02'
      IF wordlist[i] EQUALS 'SUB':
         SET wordlist[i] TO '03'
      IF wordlist[i] EQUALS 'DIV':
         SET wordlist[i] TO '04'
      IF wordlist[i] EQUALS 'SDIV':
         SET wordlist[i] TO '05'
      IF wordlist[i] EQUALS 'MOD':
         SET wordlist[i] TO '06'
```

```
IF wordlist[i] EQUALS 'SMOD':
   SET wordlist[i] TO '07'
IF wordlist[i] EQUALS 'ADDMOD':
   SET wordlist[i] TO '08'
IF wordlist[i] EQUALS 'MULMOD':
   SET wordlist[i] TO '09'
IF wordlist[i] EQUALS 'EXP':
   SET wordlist[i] TO '0A'
IF wordlist[i] EQUALS 'SIGNEXTEND':
   SET wordlist[i] TO '0B'
IF wordlist[i] EQUALS 'invalid':
   SET wordlist[i] TO '0C-0F'
IF wordlist[i] EQUALS 'LT':
   SET wordlist[i] TO '10'
IF wordlist[i] EQUALS 'GT':
   SET wordlist[i] TO '11'
IF wordlist[i] EQUALS 'SLT':
   SET wordlist[i] TO '12'
IF wordlist[i] EQUALS 'SGT':
   SET wordlist[i] TO '13'
IF wordlist[i] EQUALS 'EQ':
   SET wordlist[i] TO '14'
IF wordlist[i] EQUALS 'ISZERO':
   SET wordlist[i] TO '15'
IF wordlist[i] EQUALS 'AND':
   SET wordlist[i] TO '16'
IF wordlist[i] EQUALS 'OR':
   SET wordlist[i] TO '17'
IF wordlist[i] EQUALS 'XOR':
   SET wordlist[i] TO '18'
IF wordlist[i] EQUALS 'NOT':
   SET wordlist[i] TO '19'
IF wordlist[i] EQUALS 'BYTE':
   SET wordlist[i] TO '1A'
```

```
IF wordlist[i] EQUALS 'SHL':
   SET wordlist[i] TO '1B'
IF wordlist[i] EQUALS 'SHR':
   SET wordlist[i] TO '1C'
IF wordlist[i] EQUALS 'SHR':
   SET wordlist[i] TO '1D'
IF wordlist[i] EQUALS 'SHR':
   SET wordlist[i] TO '1E-1F'
IF wordlist[i] EQUALS 'KECCAK256':
   SET wordlist[i] TO '20'
IF wordlist[i] EQUALS 'SHA3':
   SET wordlist[i] TO '20'
IF wordlist[i] EQUALS 'invalid':
   SET wordlist[i] TO '21-2F'
IF wordlist[i] EQUALS 'ADDRESS':
   SET wordlist[i] TO '30'
IF wordlist[i] EQUALS 'BALANCE':
   SET wordlist[i] TO '31'
IF wordlist[i] EQUALS 'ORIGIN':
   SET wordlist[i] TO '32'
IF wordlist[i] EQUALS 'CALLER':
   SET wordlist[i] TO '33'
IF wordlist[i] EQUALS 'CALLVALUE':
   SET wordlist[i] TO '34'
IF wordlist[i] EQUALS 'CALLDATALOAD':
   SET wordlist[i] TO '35'
IF wordlist[i] EQUALS 'CALLDATASIZE':
   SET wordlist[i] TO '36'
IF wordlist[i] EQUALS 'CALLDATACOPY':
   SET wordlist[i] TO '37'
IF wordlist[i] EQUALS 'CODESIZE':
   SET wordlist[i] TO '38'
IF wordlist[i] EQUALS 'CODECOPY':
   SET wordlist[i] TO '39'
```

```
IF wordlist[i] EQUALS 'GASPRICE':
   SET wordlist[i] TO '3A'
IF wordlist[i] EQUALS 'EXTCODESIZE':
   SET wordlist[i] TO '3B'
IF wordlist[i] EQUALS 'EXTCODECOPY':
   SET wordlist[i] TO '3C'
IF wordlist[i] EQUALS 'RETURNDATASIZE':
   SET wordlist[i] TO '3D'
IF wordlist[i] EQUALS 'RETURNDATACOPY':
   SET wordlist[i] TO '3E'
IF wordlist[i] EQUALS 'EXTCODEHASH':
   SET wordlist[i] TO '3F'
IF wordlist[i] EQUALS 'BLOCKHASH':
   SET wordlist[i] TO '40'
IF wordlist[i] EQUALS 'COINBASE':
   SET wordlist[i] TO '41'
IF wordlist[i] EQUALS 'TIMESTAMP':
   SET wordlist[i] TO '42'
IF wordlist[i] EQUALS 'NUMBER':
   SET wordlist[i] TO '43'
IF wordlist[i] EQUALS 'PREVRANDAO':
   SET wordlist[i] TO '44'
IF wordlist[i] EQUALS 'GASLIMIT':
   SET wordlist[i] TO '45'
IF wordlist[i] EQUALS 'CHAINID':
   SET wordlist[i] TO '46'
IF wordlist[i] EQUALS 'SELFBALANCE':
   SET wordlist[i] TO '47'
IF wordlist[i] EQUALS 'BASEFEE':
   SET wordlist[i] TO '48'
IF wordlist[i] EQUALS 'invalid':
   SET wordlist[i] TO '49-4F'
IF wordlist[i] EQUALS 'POP':
   SET wordlist[i] TO '50'
```

```
IF wordlist[i] EQUALS 'MLOAD':
   SET wordlist[i] TO '51'
IF wordlist[i] EQUALS 'MSTORE':
   SET wordlist[i] TO '52'
IF wordlist[i] EQUALS 'MSTORE8':
   SET wordlist[i] TO '53'
IF wordlist[i] EQUALS 'SLOAD':
   SET wordlist[i] TO '54'
IF wordlist[i] EQUALS 'SSTORE':
   SET wordlist[i] TO '55'
IF wordlist[i] EQUALS 'JUMP':
   SET wordlist[i] TO '56'
IF wordlist[i] EQUALS 'JUMPI':
   SET wordlist[i] TO '57'
IF wordlist[i] EQUALS 'PC':
   SET wordlist[i] TO '58'
IF wordlist[i] EQUALS 'MSIZE':
   SET wordlist[i] TO '59'
IF wordlist[i] EQUALS 'GAS':
   SET wordlist[i] TO '5A'
IF wordlist[i] EQUALS 'JUMPDEST':
   SET wordlist[i] TO '5B'
IF wordlist[i] EQUALS 'invalid':
   SET wordlist[i] TO '5C-5F'
IF wordlist[i] EQUALS 'PUSH1':
   SET wordlist[i] TO '60'
IF wordlist[i] EQUALS 'PUSH2':
   SET wordlist[i] TO '61'
IF wordlist[i] EQUALS 'PUSH3':
   SET wordlist[i] TO '62'
IF wordlist[i] EQUALS 'PUSH4':
   SET wordlist[i] TO '63'
IF wordlist[i] EQUALS 'PUSH5':
   SET wordlist[i] TO '64'
```

```
IF wordlist[i] EQUALS 'PUSH6':
   SET wordlist[i] TO '65'
IF wordlist[i] EQUALS 'PUSH7':
   SET wordlist[i] TO '66'
IF wordlist[i] EQUALS 'PUSH8':
   SET wordlist[i] TO '67'
IF wordlist[i] EQUALS 'PUSH9':
   SET wordlist[i] TO '68'
IF wordlist[i] EQUALS 'PUSH10':
   SET wordlist[i] TO '69'
IF wordlist[i] EQUALS 'PUSH11':
   SET wordlist[i] TO '6A'
IF wordlist[i] EQUALS 'PUSH12':
   SET wordlist[i] TO '6B'
IF wordlist[i] EQUALS 'PUSH13':
   SET wordlist[i] TO '6C'
IF wordlist[i] EQUALS 'PUSH14':
   SET wordlist[i] TO '6D'
IF wordlist[i] EQUALS 'PUSH15':
   SET wordlist[i] TO '6E'
IF wordlist[i] EQUALS 'PUSH16':
   SET wordlist[i] TO '6F'
IF wordlist[i] EQUALS 'PUSH17':
   SET wordlist[i] TO '70'
IF wordlist[i] EQUALS 'PUSH18':
   SET wordlist[i] TO '71'
IF wordlist[i] EQUALS 'PUSH19':
   SET wordlist[i] TO '72'
IF wordlist[i] EQUALS 'PUSH20':
   SET wordlist[i] TO '73'
IF wordlist[i] EQUALS 'PUSH21':
   SET wordlist[i] TO '74'
IF wordlist[i] EQUALS 'PUSH22':
   SET wordlist[i] TO '75'
```

```
IF wordlist[i] EQUALS 'PUSH23':
   SET wordlist[i] TO '76'
IF wordlist[i] EQUALS 'PUSH24':
   SET wordlist[i] TO '77'
IF wordlist[i] EQUALS 'PUSH25':
   SET wordlist[i] TO '78'
IF wordlist[i] EQUALS 'PUSH26':
   SET wordlist[i] TO '79'
IF wordlist[i] EQUALS 'PUSH27':
   SET wordlist[i] TO '7A'
IF wordlist[i] EQUALS 'PUSH28':
   SET wordlist[i] TO '7B'
IF wordlist[i] EQUALS 'PUSH29':
   SET wordlist[i] TO '7C'
IF wordlist[i] EQUALS 'PUSH30':
   SET wordlist[i] TO '7D'
IF wordlist[i] EQUALS 'PUSH31':
   SET wordlist[i] TO '7E'
IF wordlist[i] EQUALS 'PUSH32':
   SET wordlist[i] TO '7F'
IF wordlist[i] EQUALS 'DUP1':
   SET wordlist[i] TO '80'
IF wordlist[i] EQUALS 'DUP2':
   SET wordlist[i] TO '81'
IF wordlist[i] EQUALS 'DUP3':
   SET wordlist[i] TO '82'
IF wordlist[i] EQUALS 'DUP4':
   SET wordlist[i] TO '83'
IF wordlist[i] EQUALS 'DUP5':
   SET wordlist[i] TO '84'
IF wordlist[i] EQUALS 'DUP6':
   SET wordlist[i] TO '85'
IF wordlist[i] EQUALS 'DUP7':
   SET wordlist[i] TO '86'
```

```
IF wordlist[i] EQUALS 'DUP8':
   SET wordlist[i] TO '87'
IF wordlist[i] EQUALS 'DUP9':
   SET wordlist[i] TO '88'
IF wordlist[i] EQUALS 'DUP10':
   SET wordlist[i] TO '89'
IF wordlist[i] EQUALS 'DUP11':
   SET wordlist[i] TO '8A'
IF wordlist[i] EQUALS 'DUP12':
   SET wordlist[i] TO '8B'
IF wordlist[i] EQUALS 'DUP13':
   SET wordlist[i] TO '8C'
IF wordlist[i] EQUALS 'DUP14':
   SET wordlist[i] TO '8D'
IF wordlist[i] EQUALS 'DUP15':
   SET wordlist[i] TO '8E'
IF wordlist[i] EQUALS 'DUP16':
   SET wordlist[i] TO '8F'
IF wordlist[i] EQUALS 'SWAP1':
   SET wordlist[i] TO '90'
IF wordlist[i] EQUALS 'SWAP2':
   SET wordlist[i] TO '91'
IF wordlist[i] EQUALS 'SWAP3':
   SET wordlist[i] TO '92'
IF wordlist[i] EQUALS 'SWAP4':
   SET wordlist[i] TO '93'
IF wordlist[i] EQUALS 'SWAP5':
   SET wordlist[i] TO '94'
IF wordlist[i] EQUALS 'SWAP6':
   SET wordlist[i] TO '95'
IF wordlist[i] EQUALS 'SWAP7':
   SET wordlist[i] TO '96'
IF wordlist[i] EQUALS 'SWAP8':
   SET wordlist[i] TO '97'
```

```
IF wordlist[i] EQUALS 'SWAP9':
   SET wordlist[i] TO '98'
IF wordlist[i] EQUALS 'SWAP10':
   SET wordlist[i] TO '99'
IF wordlist[i] EQUALS 'SWAP11':
   SET wordlist[i] TO '9A'
IF wordlist[i] EQUALS 'SWAP12':
   SET wordlist[i] TO '9B'
IF wordlist[i] EQUALS 'SWAP13':
   SET wordlist[i] TO '9C'
IF wordlist[i] EQUALS 'SWAP14':
   SET wordlist[i] TO '9D'
IF wordlist[i] EQUALS 'SWAP15':
   SET wordlist[i] TO '9E'
IF wordlist[i] EQUALS 'SWAP16':
   SET wordlist[i] TO '9F'
IF wordlist[i] EQUALS 'LOG0':
   SET wordlist[i] TO 'A0'
IF wordlist[i] EQUALS 'LOG1':
   SET wordlist[i] TO 'A1'
IF wordlist[i] EQUALS 'LOG2':
   SET wordlist[i] TO 'A2'
IF wordlist[i] EQUALS 'LOG3':
   SET wordlist[i] TO 'A3'
IF wordlist[i] EQUALS 'LOG4':
   SET wordlist[i] TO 'A4'
IF wordlist[i] EQUALS 'invalid':
   SET wordlist[i] TO 'A5-EF'
IF wordlist[i] EQUALS 'CREATE':
   SET wordlist[i] TO 'F0'
IF wordlist[i] EQUALS 'CALL':
   SET wordlist[i] TO 'F1'
IF wordlist[i] EQUALS 'CALLCODE':
   SET wordlist[i] TO 'F2'
```

```
            IF wordlist[i] EQUALS 'RETURN':
               SET wordlist[i] TO 'F3'
            IF wordlist[i] EQUALS 'DELEGATECALL':
               SET wordlist[i] TO 'F4'
            IF wordlist[i] EQUALS 'CREATE2':
               SET wordlist[i] TO 'F5'
            IF wordlist[i] EQUALS 'invalid':
               SET wordlist[i] TO 'F6-F9'
            IF wordlist[i] EQUALS 'STATICCALL':
               SET wordlist[i] TO 'FA'
            IF wordlist[i] EQUALS 'invalid':
               SET wordlist[i] TO 'FB-FC'
            IF wordlist[i] EQUALS 'REVERT':
               SET wordlist[i] TO 'FD'
            IF wordlist[i] EQUALS 'INVALID':
               SET wordlist[i] TO 'FE'
            IF wordlist[i] EQUALS 'SELFDESTRUCT':
               SET wordlist[i] TO 'FF'
            IF "Unknown" IN wordlist[i]:
               SET wordlist[i] TO wordlist[i][1:3]
            i += 1


      #save opcodehexadecimal output IN .txt format
   opcodeoutputhexa="C:\\Users\\user\\Desktop\\opcodehex\\opcodehex"+str(lline)+".txt"
      with open(opcodeoutputhexa, 'w') as WriteOpcodeOutputHexa:
         WriteOpcodeOutputHexa.write(' '.join([''.join(l2) FOR l2 IN wordlist]))


Inserting the opcode results in CSV File

      # Open the text file and read the data
      with open(opcodeoutputhexa, 'r') as ReadOpcodeOutputHexa:
         SET Read_Hexa_Opcode TO ReadOpcodeOutputHexa.read()

      # Open the CSV file IN read mode
      with open('Final_Dataset.csv', 'r') as csv_file:
         SET reader TO csv.reader(csv_file)
```

```
    # read the data
    SET data TO [row FOR row IN reader]

# update the data
SET data[countexcel][2] TO Read_Hexa_Opcode

# Open the CSV file IN write mode
with open('Final_Dataset.csv', 'w', newline='') as csv_file:
    SET writer TO csv.writer(csv_file)

    # write the updated data
    writer.writerows(data)
```