Finding Vulnerabilities in Smart Contracts

Michele Massetti
big.michelemassetti@gmail.com
Karlsruhe Institute of Technologies
Karlsruhe, Baden-Württemberg, Germany

ABSTRACT

Blockchain is a revolutionary technology that enables users to communicate in a trustless manner. The most prominent change brought by this technology is the mode of business between organizations: they do not need anymore a trusted third party. It is a distributed ledger technology based on a decentralized P2P (peer-to-peer) network. Since Bitcoin was deployed, many blockchain systems have been born with more capabilities, which have allowed them to fit many different use cases. Smart Contracts, which are programs running on blockchain systems, could extend the potentiality of blockchain from a platform for financial transactions to an all-purpose utility. The development of innovative and prominent applications is a consequence of them, such as NFTs (non fungible token) marketplaces, music royalty tracking, supply chain and logistics monitoring, voting mechanisms, cross-border payments, and many others. Finding bugs and vulnerabilities in them is necessary for assuring their correct behaviour. This paper deals with the way for finding the vulnerabilities in Ethereum-based Smart Contracts. We review related work regarding the classification of the most common vulnerabilities and tools which support their detection.

KEYWORDS

Solidity, Software, Vulnerability, Blockchain

1 INTRODUCTION

Ethereum is the most used envirimont regarding smart contracts. Its DeFi (decentralized finance) and dApps (decentralized applications) count the highest amount of dollars exchanged rather than the other distributed systems. Moreover, it can be described as the "internet of Blockchain", because Ethereum allows other systems to be built on top of itself, adopting its technology. Its ecosystem consists of a large variety of smart contracts deployed on-chain, and a wide range of valuable assets.

This growing technology has fascinated many investors, indeed, according to CoinGeko, the crypto market's value is standing around $2 trillion. On the other hand, interest in such a market has grown even among malicious attackers. Attacks such as the "Parity Wallet Hack" and the "Decentralized Autonomous Organization Attack" cost millions of dollars simply because of naive bugs in the smart contract code. Blockchain and smart contract technologies have multiple aims, but unfortunately, new applications based on them still contain bugs and multiple vulnerabilities, which cause several issues for the end-users. Most of the use of this technology relates to finance or certifications, therefore integrity, authentication and authorisation in transactions are mandatory.

The research field behind blockchain technology is growing, as well as the one concerning its security and accordingly, many analysis tools were developed. These incorporate various strategies for performing the analyses, concerning the technical aspects of smart contracts, so these would work differently according to the object of the analysis.

Among the many aspects of smart contract, our systematic literature review focuses on studies dealing with vulnerabilities and methodologies, and the tools which implemnt those ones, for their detection. We will try to give an answer to the following research questions:

- Which are the main vulnerabilities in Smart Contracts?
- Which methodologies are implemented by analysis tools and which are the most used ones?
- Which are possible new approaches for discovering vulnerabilities?

Section 3 gives an overview of blockchain and smart contracts technology, the execution environment and model. In our work we focus on the Ethereum platform, however many concepts can be adapeted in multiple blockchain EVM (Ethereum Virtual Machine) compatible.

In Section 2, we deal with the actual papers and studies regarding this topic, citing the ones we reviewed for the writing of this paper.

Section 4 gives a taxonomy for the main vulnerabilities about Solidity.

A possible classification of analysis tools is drawn in Section 5. Moreover, their strategies are explained and it covers even possible ways for vulnerabilities detection.

Section 6 is addressed to a comparison between the discussed methodologies and possible strategies for the prevention of exploits.

2 RELATED WORK

Our work is addressed to define the most common vulnerabilities and the most used tools for finding those. The first step was getting in touch with research works which deal with topics such as vulnerabilities and analysis tools. For a better comprehension of the matters, we collect information about the basis of blockchain technology and security terminology. [5] provides an overview of Ethereum environment.

We picked the analaysis tools strating from the reading of surveys and papers as [2], [14], [15], [16], [9]. Those deal with a great number of tools, defining their characteristics and those produce a comparison between tools based on selected benchmarks. We deepen our knowledge about the tools by reading their provided papers ([3] [6] [12] [17] [11]).

Colloquium on Vulnerability Discovery 2022, August 1st, 2022, KIT, Karlsruhe, Germany

[8] compare different strategies implemented by tools, citing even Machine Learning, which has been used in this matter just recently. Since Machine Learning popularity has grown in the computer science, recent papers has proposed it as a technique in this field, for example [10], which tries to combine it with fuzzing, and [18].

We drew the definition of the vulnerabilities through the benchmark used by the papers, moreover, we based our selection on the two most used open-source databases for smart contract vulnerabilities, which are [13] and [1].

3 BACKGROUND

This section gives to the reader some basic knowledge about blockchain, focusing on Ethereum and its smart contracts. We briefly introduce its general structure and how security is applied in this context.

3.1 Blockchain

Blockchain is a shared ledger that holds transactions and information in a decentralised peer-to-peer network. A blockchain acts as a database, storing information in a digital format. This technology is best known for its vital role in cryptocurrency systems, such as Bitcoin, to maintain a secure and decentralized record of transactions. The innovation is that it ensures the truthfulness and security of data records and creates trust without the need for a trusted third party.

It acts like a database, however, it organises data into units called blocks, each of which contains a collection of data. Blocks have specific storage capacities, and when they are full, they are closed and linked to the preceding block, producing a data chain known as the blockchain. All additional information added after that newly added block is compiled into a new block, which is then added to the chain after it is filled.

Each computer in the network must agree on each new block and the entire chain. These calculators are called "nodes". These ensure that everyone who interacts with the blockchain has the same data. New blocks are broadcasted to network nodes, which track and verify those, moreover, the state of the blockchain is updated for everyone. To achieve this distributed agreement, blockchains need a consensus mechanism.

The purpose is to allow digital information to be recorded and distributed, but not edited. In this way, an immutable ledger or record of transactions is generated, thus that cannot be modified, deleted, or destroyed. This is why blockchain is also known as DLT (Distributed Ledger Technology).

3.2 Ethereum Environment

Ethereum is a blockchain-based global decentralised software platform. Anyone can use Ethereum to construct any secure digital technology they can imagine. It has a token built for usage on the blockchain network, but it may also be used to pay for work done on the blockchain by participants. It is built to be decentralised, scalable, programmable, and secure. It is the blockchain of choice for developers and businesses who are building technologies on top of it to transform numerous sectors and our everyday lives. It natively supports smart contracts, which are the essential tool behind decentralized applications. Regarding development and interaction, Ethereum is the leader in this sector.

In the Ethereum universe, there is a single, canonical computer (known as the Ethereum Virtual Machine, or EVM) whose state everyone on the Ethereum network agrees on. Every Ethereum node (participant in the Ethereum network) saves a copy of this computer's current state. A participant can also broadcast a request for this computer to conduct any computation they choose. When a request like this is broadcast, other network members verify, validate, and carry out (or "execute") the computation. The EVM's state changes as a result of this execution, which is committed and propagated throughout the network. The consensus algorithm, which ensures the correct system's behaviour, is called "proof of works". It involves that anyone, who wishes to add additional bricks to the chain, must first solve a complex challenge that takes a lot of processing resources. Solving the puzzle "proves" that you used computing resources to do the "job." Mining is the process of doing so. It is usually done by trial and error, but adding a block earns you ETH. The nodes, which compute these types of operation, are called "miners".

Recently, Ethereum is moving off of POW, it is implementing "proof of stake" as a consensus algorithm. It has the consequence that the nodes are no anymore called miners but validators. These have to stake, and collect, at least 32 ETH on a smart contract on the chain. The nodes have the same role of validating the transaction, but in this way, computational and electrical energies are saved.

Transactions are cryptographically signed instructions from accounts, which are any entities which can collect the native token, called ETH or ether. An account will initiate a transaction to update the state of the Ethereum network. The miners are responsible to collect the transaction, changing the state of the EVM and broadcasting those in the network. Transactions require a fee and must be mined to become valid. The unit of measurement for the amount of computing effort necessary to perform various activities on the Ethereum network is called gas. It represents the fuel of the blockchain. The account can set a limit on the amount of ETH to pay. If the price of gas is less than the one paid by the user, it has a refund. If the amount of fees is not enough, a "out of gas" exception can occur.

3.3 Smart Contracts and Solidity

The definition given by the Ethereum foundation of Smart concract is:

A "smart contract" is simply a program that runs on the Ethereum blockchain. It's a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

It is considered a type of account, which is any entity with an ether balance that can send transactions on Ethereum. They are deployed on the network and run according to the code. User accounts can then engage with a smart contract by sending transactions that instruct the smart contract to perform a function. Smart contracts, like conventional contracts, can set rules and have them enforced automatically through programming. They can't be deleted by default, and their interactions are permanent.

The EVM smart contracts can be programmed in several languages. The most active and maintained language is Solidity, which our work deals with.

It is an object-oriented and high-level language. It supports:

- inheritance;
- libraries;
- complex user-defined types.

A smart contract, as a program developed in other programming languages, contains parameters, a constructor and functions. A peculiarity of this language is the fallback functions.

They have been involved in many attacks, as will be explained in section 4, but they were developed without a malicious aim. They are external functions with neither a name, parameters, or return values. Only one fallback can be assigned to a contract. A fallback function is triggered:

- if a function identifier does not match any of the available functions in a smart contract;
- if there was no data supplied along with the function call.

A fallback function always receive data, but it should be marked as payable for accepting ETHs.

3.4 Security of Smart Contracts

Smart contracts are public and available by every account on the chain, but they can represent a drawback. In the security field, developers can implement some strategies by design for preventing malicious operations. For protecting software, multiple layers of secuirty can be implemented: a firewall can be deployed or even an IDS (intrusion detection system), the software can be achieved just by logging in to the VPN (virtual private network) scanning all the traffic.

On the other hand, none of those can be applied in smart contracts environment, so finding vulnerabilities is a big challenge for developers. Malicious accounts can exploit the bug inside the smart contract, but developers should be aware the order of the execution of transactions depends even on the miners which compute the blocks.

4 VULNERABILITIES IN SOLIDITY

Related taxonomies by scholars and open-source projects are the starting point for drawing our classification of Vulnerabilities in Solidity. We selected the most recurrent vulnerabilities that we found during our research. The term vulnerability is used in a similar sense than is common in computer security. It refers to a weakness or limitation of a smart contract that may result in security problems. A vulnerability represents even the way the attacker can exploit the contract. This includes locked or stolen resources, breaches of confidentiality or data integrity, and state changes in the environment of smart contracts that were not detected by developers or users, therefore they possibly are exploited by a malicious part.

We summaries different vulnerabilities, providing a definition and Table 1 shows the correlation with Dasp10 and SWC, associating per each vulnerability the identifier per project.

The last section deals with an overview of bugs in the logic of a smart contracts. It awarns that a vulnerability can rely even upon an incorrect implementation of the contracts.

4.1 Reentrancy

Reentrancy is one of the vulnerabilities much more exploited in recent years. Many attacks occurred and could steal a great amount of money from vulnerable protocols. A reentrancy attack can occur when a function makes an external call to another untrusted contract before it resolves any effects and the normal execution flow of the program is modified. A recursive call back to the original function is made by the attacker, before the function could update the state of the variables. The goal is repeating interactions that would have otherwise not run after the effects were resolved. The function withdraws, lines 5 to 11 Listing 1, implements this bug, because the balance of the user is updated after the external call. The attacker, with the usage of a malicious smart contract, can call again the vulnerable function. It is run multiple times, without updating the balance, and withdrawing all the funds. There are several countermeasures for this type of bug: semaphores logic can be implemented with the modifiers, and the update of the variable can be moved before the external call. The external call can be safe if the developer checks the amount of gas used, to avoid an attacker can modify the control flow. One of the fundamental of this attack is the usage of fallback by the malicious contract, which is triggered every time it receives ETH.

```
1  contract EtherStore {
2      mapping(address => uint) public balances;
3      ...
4      //vulnerable function exploited by the attacker
5      function withdraw() public {
6          uint bal = balances[msg.sender];
7          require(bal > 0);
8          (bool sent, ) = msg.sender.call{value: bal}("");
9          require(sent, "Failed to send Ether");
10         balances[msg.sender] = 0;
11     }
12 }
13
14 contract Attack {
15     constructor(address _etherStoreAddress) {
16         etherStore = EtherStore(_etherStoreAddress);
17     }
18
19     // Fallback is called when EtherStore sends Ether to
           this contract.
20     fallback() external payable {
21         if (address(etherStore).balance >= 1 ether) {
22             etherStore.withdraw();
23         }
24     }
25     function attack() external payable {
26         require(msg.value >= 1 ether);
27         etherStore.deposit{value: 1 ether}();
28         etherStore.withdraw();
29     }
30
31 }
```

Listing 1: Reentrancy

Table 1: Collection of Vulnerabilities

Name	DASP10 DB	SWC DB
Reentrancy	DASP-1	SWC-107
Unexpected Ether balance		SWC-132
Overflows & Underflows	DASP-3	SWC-101
Block Timestamp Manipulation	DASP-8	
Transaction Ordering Dependence	DASP-7	SWC-114
Denial Of Service (DOS)	DASP-5	SWC-113
Out of Gas	DASP-5	SWC-128
Unchecked CALL/SEND Return Values	DASP-4	SWC-104

4.2 Unexpected Ether balance

A smart contract of which logic relies on its balance can be exposed to unexpected receiving of ETH. A DOS attack can occur when the smart contract strictly assumes a specific token balance. In Solidity there is the fallback called "receive", which is a function triggered by the reception of ETH, so the developer can prevent a transaction could send ether. However, it is always possible to forcibly send ether to a contract (without triggering its fallback function), using "self-destruct", or by mining to the account. Smart Contracts which rely on code execution for every ether sent to the contract can be vulnerable to attacks where ether is sent. In the provided example (Listing 2), in the function play() line 5 there is a require, which checks no players can play when the balance reaches the predefined threshold. The function claimReward(), allows the user to withdraw when the balance reaches the threshold. An attacker can send ether to this contract, for manipulating its balance. If it was greater than the threshold, the functions would always revert because of the require. As a consequence, no one could withdraw the funds and call the function play().

```
1   ...
2   function play() public payable {
3          require(msg.value == 0.5 ether); // each play is
               0.5 ether
4          uint currentBalance = this.balance + msg.value;
5          // ensure no players after the game as finished
6          require(currentBalance <= finalMileStone);
7          // if at a milestone credit the players account
8          if (currentBalance == payoutMileStone1) {
9              redeemableEther[msg.sender] +=
                   mileStone1Reward;
10         }
11         else if (currentBalance == payoutMileStone2) {
12             redeemableEther[msg.sender] +=
                   mileStone2Reward;
13         }
14         else if (currentBalance == finalMileStone ) {
15             redeemableEther[msg.sender] += finalReward;
16         }
17         return;
18  }
19  function claimReward() public {
20    // ensure the game is complete
21    require(this.balance == finalMileStone);
22    // ensure there is a reward to give
23    require(redeemableEther[msg.sender] > 0);
24    redeemableEther[msg.sender] = 0;
25    msg.sender.transfer(redeemableEther[msg.sender]);
26  }
27    ...
```

Listing 2: Unexpected Ether

4.3 Overflows & Underflows

The Ethereum Virtual Machine (EVM) defineds fixed-size data types for integers. An example is the identifier uint8, unsign integer over 8 bit, for variables means the variable can only store numbers in the range [0,255]. The attacker can craft the input value of a vulnerable contract's function in order to force a specific operation. In the contract TimeLock in Listing 3, an attacker could forge a parameter of the function function increaseLockTime, in oder to set lockTime to 0 because of Overflows. So it can call withraw and passing the require at line 17. The most common way for solving these issues is to build or use mathematical libraries which replace the standard math operators. OppenZepplin is a company, dealing with Blockchain technologies. It has done a great job in building and auditing secure libraries which can be leveraged by the Ethereum community. SafeMath library can be found in most of the deployed smart contracts.

In Solidity version 0.8.0 arithmetic operations revert on underflow and overflow, consequently this vulnerability is avoided.

```
1   contract TimeLock {
2       mapping(address => uint) public balances;
3       mapping(address => uint) public lockTime;
4       function deposit() public payable {
5           balances[msg.sender] += msg.value;
6           lockTime[msg.sender] = now + 1 weeks;
7       }
8       function increaseLockTime(uint _secondsToIncrease)
             public {
9           lockTime[msg.sender] += _secondsToIncrease;
10      }
11      function withdraw() public {
12          require(balances[msg.sender] > 0);
13          require(now > lockTime[msg.sender]);
14          msg.sender.transfer(balances[msg.sender]);
15          balances[msg.sender] = 0;
16      }
17  }
```

Listing 3: Overflows & Underflows

4.4 Block Timestamp Manipulation

Miners are responsible for collect validate the transaction, stored in blocks and deployed on-chain. They can manipulate the timestamp of the block that they are managing and influence the execution of the transactions.

A miner can forge the timestamp by a few seconds, for obtaining benefits. The timestamp dependence vulnerability occurs from a flawed comprehension of timekeeping. It enables the Ethereum network to be detached from the synchronized global clock. The block timestamp can be used as entropy for random numbers, locking funds for periods of time and various state-changing conditional statements that are time-dependent. Since the smart contract permits miners to put up a timestamp within 30 seconds of block validation, this gives a miner more opportunity for exploitation. Listing 4 shows a function (a fallback) which implements the logic of a lottery. The users can send ether to this contract, which accepts just one transaction per block. If the block is a multiple of 15, the user receives all the funds. This function is based on the timestamp of the block, A malicious miner could adjust the timestamp for getting the win.

```
...
// fallback function used to make a bet
function () public payable {
        require(msg.value == 10 ether); // must send 10
            ether to play
        require(now != pastBlockTime); // only 1
            transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
}
...
```

Listing 4: Block Timestamp

4.5 Transaction Ordering Dependence

Miners are responsible for the creation and certification of the blocks on the blockchain. Inside those, the transactions are stores, but their order is not guaranteed by design and the miners can easily manipulate that. Consequently, this makes smart contracts vulnerable to transaction order dependency, which can be classified as a type of race condition. The vulnerability relies on smart contracts' critical decisions based on the order of transactions. Moreover, if two dependent transactions are invoked by the same contract then the ordering of transactions must affect the states of the blockchain. So we can have a case of race condition, even with a miner without malicious aims.

The smart contract MarketPlace (Listing 5) implements the logic of a market place. The users can buy an arbitrary amount of stock and the owner can set the price. In this case, the variable price is influenced by two functions. If the owner of the contract would be a miner, it could manipulate the order of the transaction. The miner-seller can influence the transaction orders and update the price after the buy request was sent, but before it is processed, effectively increasing the anticipated price for the buyer. The seller can just modify the price, while a user is buying and it can affect its action.

```
contract MarketPlace {
  uint256 price;
  uint256 stock;
  address owner;
  constructor() {
    owner= msg.sender;
  }

  function updatePrice(uint256 amount) public {
    require(msg.sender==owner);
    ...
  }
  function buy(uint256 amount){
    require(stock>=amount);
    require (msg.value >=amount*price);
    stock-=quant;
  }
}
```

Listing 5: Transaction Ordering Dependence

4.6 Denial Of Service (DOS)

Denial of service attack is a category of attack, which involves all those that have the aim to disturb the functionality or stop the execution of the target system. In the context of smart contracts, we provide an example of this category, which involve multiple strategies for breaking the normal flow execution of the program.

Listing 6 shows a simplified contract for auction. The function refund users() has the responsibility to refund all the bidders. The function contains a loop, so it can get the amount of the bid of the user and refund it. The bug relies on the require inside the loop. If the transaction returns false, the require fails, so all the transactions are reverted, consequently, all the previous refunds are reverted. A malicious contract can be implemented for taking part in the function, but that one reverts every time it receives ether. In this way, the function refundUsers() would always revert and no users can be refunded.

```
contract Auction {
  mapping (address=>uint256) allBids;
  mapping (uint256=>address) users;
  uint256 nUsers;
  uint256 lastrefund;
  ...
  function refundUsers() {

    for(uint256 i = lastrefund; i<nUsers;i++){
      address user=users[i];
      uint256 amount= allBids[user];
      (bool sent, )= user.call{value: amount}("");
      require(sent);
    }

  }
  ...
}
```

Listing 6: Denial of Service/Out of Gas

4.7 Out of Gas

In Ethereum, each computational operation requires a fee to be paid by the account which runs the transaction, measured in the unit called gas. In the Ethereum documentation, gas is defined as follows: “Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee.

Gas refers to the fee required to conduct a transaction on Ethereum successfully."

The sender pays the transaction fee, previously specifying the gas limit, If the paid amount exceeds the fee, it will receive a refund when the transaction finishes. If the actual execution cost of the transaction exceeds the gas limit, the transaction will fail with an out-of-gas error. We used Listing 6 for showing an example of DOS, but it implements an out-of-gas vulnerability too. The amount of gas depends on the number of the for loop's iteration. If the number of iterations grows too much, the function would always revert. A malicious attacker can manipulate the number of iterations, using a great number of wallets for the bid.

4.8 Unchecked CALL/SEND Return Values

Solidity provides multiple ways for performing an external call. The transfer() method is the most common way of exchanging ether between smart contracts. It reverts if something goes wrong and the function does not succeed. However, the send() and call() functions can also be used and, for a more versatile external call. These functions return a boolean indicating if the call succeeded or failed. A common bug relies on the return value is not checked, rather the developer expects a revert to occur.

The user can call the function withdrawEmergency() at line 4, Listing 7, for withdrawing all its funds. The function updates the value of the mapping balances, setting the balance of the user to zero. If the function does not succeed, it does not revert. The funds of the user would be stacked inside the contract. Moreover, the amount of ether in the contract is miscomputed.

```
1  contract Bank{
2    mapping (address=> uint256) balances;
3    ...
4    function withdrawEmergency() public {
5      require(balances[msg.sender] > 0);
6      uint256 amount= balances[msg.sender];
7      balances[msg.sender] = 0;
8      etherLeft -= _amount;
9      msg.sender.send(amount);
10   }
11   ...
12 }
```

Listing 7: Unchecked CALL/SEND Return Values

4.9 Vulnerable Logic

The previous vulnerabilities specifically deal with problems involving Solidity. However, the smart contract, seen just as a program, could have a weak implementation. The most frequent issue, though, is typically a simple mistake in the smart contract's logic. These mistakes, which have a significant impact on security, can be the result of simple typographical errors or specification ambiguities.

Decentralized finance (DeFi) is an emerging financial technology based on blockchain. Smart contracts usually implement the logic of a bank, providing financial operations. Since these collect a great amount of money, an error in their logic can cost millions of dollars. A leak of security can be caused by a bug dealing with the code itself (Solidity issue) or even by the logic of the program itself.

Examples of bad logic implementation are the bugs concerning the pools, with leaks which allowed the attacker to drain the liquidity. Liquidity pools, in the DeFi, are reserves of tokens secured in smart contracts. They are the primary technology underlying the present DeFi ecosystem. They allow user to deposit tokens receiving an interest on the deposit or provide lend. Bad implementation of the mechanism of reward or a miscalculation of the rewards can have the consequence of an exploit.

A reason for an attack can rely even on the miscomputation of the algebraical formulas; it could bring to an exploit, even if the implementation in Solidity would be correct.

5 METHODS FOR FINDING VULNERABILITIES

This section is addressed to present the most effective and common tools with their strategies in Ethereum smart contracts context. We classify the tools in 2 main categories: with and without specification. The tables 2 and **??** collect the tools. In the first case, tools need specifications for running. On the other hand, the former group deals with those which can give a result from the solidity file. The last subsections deals with general methodologies which are very useful in this context.

Machine Learning has been adopted in research and it is getting more effective.

Checking the absence of bugs is fundamental, but even certifying the contract fulfils the prefixed goals is necessary. Thus, before deployment on the blockchain, there are strategies to control the behaviour of the contract in a similar environment.

5.1 Tools without specifications

In this subsection we present the list of tools without specification, providing for each of them a general overview of its methodology for finding vulnerabilities.

Slither. Slither is a smart contract code static analysis framework. It has rapid and reliable security detection algorithms for potential bugs. Slither can be used for automatic vulnerability identification, automated optimization detection, code interpretation, and assisted code review, among other things. A multi-stage approach is the key point for the analysis. From the contract source code, the Solidity compiler generates a Solidity Abstract Syntax Tree (AST), which represents the input source of the tool.

Slither definents the issues of the contract in this and important features. The inheritance graph, Control-flow graph (CFG), and so on are generated during the early stage. The next stage includes converting the complete code to SlithIR. The last step covers the computation of a list of pre-defined analyses.

Mythril. Mythril is an open-source security tool that examines Solidity-based smart contracts. It employs the symbolic execution technique as methodology for the analysis. The bytecode of the smart contract is generated and then is run in a custom EVM, for looking for security issues. When a programme bug is detected, the input transactions are examined to establish the cause.

This security strategy aids in determining the root cause of a program's vulnerability as well as mitigating exploitation.

Table 2: Tools without Specifications

Name	Keywords
Slither	Static Analysis Scanner
Mythril	Symbolic execution, Contract Bytecode
SmartTest	Symbolic execution, Susceptible transaction sequences,
Oyente	Symbolic execution, Execution paths

Mythril accepts as input the source code of the contract, but even its bytecode. If the user provides the end point with the blockchain, it can analyse even contracts on-chain.

Its ouput provides the list of the transaction for triggering the bug and the label of the SWC database for defining the vullnerability.

SmartTest. SmartTest is an Ethereum smart contract safety analyzer. For effectively detecting susceptible transaction sequences, it uses a symbolic execution method. The tool to detect transaction sequences, showing the weaknesses of the examined smart contract, is the project's core problem. As a result, bugs are discovered as the source of multiple transaction interaction. SmartTest's goal is to automatically deliver vulnerable transaction sequences that demonstrate the smart contract's flaws. The fundamental idea is to leverage known vulnerable transaction sequences to develop a statistical model that can be used to drive symbolic execution towards more successfully discovering unknown vulnerabilities.

Statistical language models direct symbolic execution, allowing it to select transacion sequences that are likely to reveal vulnerabilities. The first step in this strategy is to perform unguided symbolic execution on existing vulnerable contracts, followed by learning a probability distribution over vulnerable transaction sequences. The detection of the following six types of security-critical vulnerabilities are supported by the tool: integer over/underflow, assertion violation, division-by-zero, ERC20 standard violation, Ether-leaking vulnerability (e.g., unauthorized access to transfer), and suicidal vulnerability (e.g., unauthorized access to selfdestruct).

A limitation of this tool involves the lack of the detection of vulnerabilities that require analysis of the interaction of multiple contracts to demonstrate the flaws: reentrancy is not covered.

Oyente. Oyente is a symbolic execution tool for finding smart contract security flaws, written in Python. It analyses Ethereum smart contracts for security flaws that could lead to possible attacks. Dangerous bugs are not the only focus of this, but it also looks at every possible execution path, which can trigger an exploit.

The Solidity compiler is used for obtaining bytecode, necessary for the analysis, and the disassembler from Go-Ethereum displays opcodes in symbolic form.

The symbolic execution approach uses a mathematical formula to reflect the characteristics of an execution path. Oyente compares the new formula to the ones that contain common bugs, so it can verify if both formulas are valid at the same time.

5.2 Tools with specifications

The tools with specifications are listed in this subsection. We outline their main characteristics and briefly summarise their configurations.

Echidna. Echidna, Slither and Manticore were developed by the same researchers. It's a sophisticated fuzzer for EVM that finds problems in Solidity code. To conduct deep analysis for defects, this tool just requires Solidity propositions and provides a simple user interface (UI) to simplify its results. Echidna tries several combinations of inputs until it succeeds in breaking the given property. Echidna has a few characteristics with Manticore, allowing it to function at the EVM level. It can also be merged into CI (continuous integration) to detect code issues while development is ongoing.

The 2 most effective modes of this tool are the property mode and the assertion mode. In the first case, the developer writes in the smart contract a function, called the prefix "echidna_test". It has the keywords public, callable by anyone, and view, because it does not alternate the variables. The return value is a boolean and it represents the property which the tool tries to break, for example, a variable uint cannot be greater than 100. On the other hand, in assertion mode, the developer writes the properties to check using the "assert(condition)". In both cases, the sequence of the function call for breaking the property is given, if it is found.

Since it provides the list of function call of the contracts,a limitation of this tool is its impossibility to detect a reentrancy issues, which involves external ones. It can cover those, just if the malicious contract is provided, so it recognise the sequence of function for the exploit.

Manticore. Manticore is a Solidity audit tool that analyzes smart contracts symbolically. Manti-key core's functions include tracing inputs that terminate a program, logging instruction-level implementation, and offering Python API access to its analysis engine. It features a capability called dynamic symbolic execution that examines binaries and Ethereum smart contracts. The Core Engine, Native Execution Modules, and Ethereum Execution Modules are the three main components of Manticore's architecture. Secondary features include the Satisfiability Modulo Theories (SMT-LIB) module, Event System, and API.

Manticore can be run as a verifier, which means the developer specifies properties in the contract and the tool tries to break those. It has a similar grammar to Echidna, but in this case, the function name prefix is "crytic_test". Otherwise, Manticore implements scanner modules for solidity, so it can detect the risk of typical vulnerabilities, such as reentrancy.

SolcVerify. SolcVerify is an Ethereum smart contract source-level verification tool. It accepts Solidity smart contracts and uses modular programme analysis to discharge verification conditions. Because it is built on top of the Solidity compiler, it can reason at the contract source code level. Solc-verify can reason about high-level

Table 3: Tools With Specifications

Name	Keywords
Echidna	Fuzzing, security properties and assertion analysis
Manticore	Symbolic Execution, security properties analysis
SolcVerify	Formal Verification

contract properties while reliably modelling low-level language semantics as a result of this. It is a compiler extension for the Solidity language. It receives a collection of Solidity contracts, including specification annotations, and discharges verification conditions using the Boogie verifier and SMT solvers. It converts annotated contracts to Boogie Intermediate Verification Language (BIVL).

5.3 Machine Learning

The adoption of machine learning techniques in the field of smart contracts is a result of their growing popularity. To identify suicidal, prodigal, and greedy contracts, [19] implemnts a long-short term memory (LSTM). By classifying contracts that were discovered by MAIAN as vulnerable during training, they were able to improve the LSTM model's test accuracy.

[7] deals with the training of a CNN (convolutional Neural Network) by converting a sizable amount of Ethereum bytecode into RGB colour coding, trained a CNN model. The model now automatically analyses input bytecode to flag potential compiler errors.

ContractWard ([20]) is an example of attempt of Automated Vulnerability Detection Models for Ethereum Smart Contracts. It can detect the following six types of vulnerability:

- integer Overflow and Integer Underflow Vulnerabilities;
- transaction-Ordering Dependence (TOD);
- callstack depth attack vulnerability;
- timestamp dependency;
- reentrancy vulnerability.

The model adopts two simple classification algorithms, namely, SVM and KNN together with two sampling methods. The source codes of the smart contracts selected as datasets are transformed into operation codes (opcodes). Oyente is used to label all the contracts, which have six labels each. N-Gram, which refers to n words that appear continuously in text, is the algorithm implied for feature extraction. One vs. Rest (OvR) algorithm is adopted for multi-label classification where C1, C2, C3, C4, C5 and C6 correspond to the six selected vulnerabilities. Finally, detection models are built based on the balanced training sets for the detection.

5.4 Testing Smart Contracts On-Chain

The purpose of the analysis tools is to provide assurance that there are no bugs. Some of them even attempt to violate the provided rules or check ones. However, developers even have the ability to test the contract logic in a similar to the actual environment, to verify the correct program workflow.

The deployment of a smart contract in an environment as much similar to the on-chain one, helps the programmers to test and have a response to the behave of the software.

Multiple blockchain systems provide a "test-net". A test-net is used for testing and experimentation without risk to real funds or the main chain. It is the same behaviour as the main-net, but then it has a different purpose, for example, the number of native tokens is unlimited. About Ethereum, Ethereum Kovan is the most used one.

The network can be virtualised on a machine for development. Hardhat is a powerful tool which carries that. As presented in their documentation ([?]), Hardhat comes built-in with Hardhat Network, a local Ethereum network node designed for development. It provides the option to fork the main-net, so it is possible to reproduce exactly the real on-chain state. It allows to fork the chain, starting from a specific block.

6 DISCUSSION

Since every user on the blockchain can access directly to the smart contracts, the security of those is a challenge. The source code might be private, but it is always possible to draw the bytecode; in most projects, for ingratiating the investors, developers publish the source code of those.

A in-depth approach to scanning bugs is necessary before the deployment. The tools without specification can be very effective in the most common vulnerabilities, such as reentrancy. On the other hand, recently many attacks involve the logic of the program, such as the estimation of rewards for a holder, or the creation of a taken. Such similar vulnerabilities cannot be gotten by those, because in this case the property is defined by the developer itself. Tools with specifications come in our help, in this case, allowing the tester to check if the property can be broken by a malicious sequence of functions. Multiple approaches have been adapted for smart contracts, but the research is still investigating for adapting more valuable analysis techniques.

ML (Machine Learning) has been used for the analysis of transactions and for defining specific aspects on-chain. Another prominent use is the financial one, for estimating the best time for computing a transaction through data analysis.

However, ML applied to the scanning of solidity smart contracts research is just at the beginning. Since the amount of data is growing, there are more possibilities that the accuracy of the modules can increase. An example of ML applied in the security field is a malware scanner, which firstly converts the files as coloured images and then applied the Neural Nets to that. The problem with smart contracts is the small number of lines of code which handle a great number of investments. A misposition of a line of code might bring a loss of millions of dollars. Therefore, a false positive can have disastrous consequences.

For having a complete overview of the behaviour of a smart contract, it should be deployed in an environment as similar as possible to the main net. Most of the tools do not allow inheriting specific information from the main-net and they should be set up correctly. The possibility of launching on a test-net, or virtual environment, allows the developer to have a deepen view of the effectiveness of the contract. Many tools allow to write test even in different languages, as javascript, and provide useful information as the code coverage.

The high risk of attacks on the blockchain as a consequence of this technology is still in its first steps. An absence of legal regulation contributes to the increasing number of cyber crimes in this context. The adoption of this technology in more official fields, such as the adoption of a cryptocurrency as the currency of a country, would contribute to the research in security.

7 CONCLUSION

Users can create decentralised digital agreements using smart contract technology without the aid of a third party. Sectors such as health, company management, shareholder agreements, and insurance were drawn to this technology.

However, as this technology's adoption has increased, potential attackers get more interested in it, which leads to several serious exploitations. Multiple security strategies adopted in common programs can be used in this context, but smart contracts logic has peculiarities.

In this paper, our research presented the most common vulnerabilities in Solidity smart contracts. Those can be written even with different programming languages on different blockchains, but Solidity represents the most used one; some EVM-compatible ones that mostly adopt Solidity as the language for smart contracts are Avalanche, Polygon, IOTA, Tendermint, and Binance Smart Chain.

We selected and presented some analysis tools, describing their strategies. More security approaches can be applied in this field and the research is growing. Machine learning is still in the research phase, but in the next years, we expect the development of multiple tools with this technology.

REFERENCES

[1] Mason Hemmel David Wong. 2018. Decentralized Application Security Project (or DASP) Top 10. https://www.dasp.co/.

[2] Monika di Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 69–78. https://doi.org/10.1109/DAPPCON.2019.00018

[3] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. *CoRR* abs/1908.09878 (2019). arXiv:1908.09878 http://arxiv.org/abs/1908.09878

[4]]Hardhat Ethereum Foundation. [n. d.]. Hardhat Documentation. https://hardhat.org/

[5] Ethereum Foundation. 2020. Ethereum developer resources. https://ethereum.org/en/developers/docs/

[6] Will Song Artur Cygan Josselin Feist Alex Groce Gustavo Grieco, Trail of Bits. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. (2020).

[7] TonTon Hsien-De Huang. 2018. Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks. *CoRR* abs/1807.01868 (2018). arXiv:1807.01868 http://arxiv.org/abs/1807.01868

[8] Shinhae Kim and Sukyoung Ryu. 2020. Analysis of Blockchain Smart Contracts: Techniques and Insights. In *2020 IEEE Secure Development (SecDev)*. 65–73. https://doi.org/10.1109/SecDev45635.2020.00026

[9] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* 10 (2022), 6605–6621. https://doi.org/10.1109/ACCESS.2021.3140091

[10] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. 2019. SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 458–465. https://doi.org/10.1109/IOTSMS48152.2019.8939256

[11] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 16 pages. https://doi.org/10.1145/2976749.2978309

[12] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019). arXiv:1907.03890 http://arxiv.org/abs/1907.03890

[13] MythX Project. 2020. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/.

[14] Heidelinde Rameder. 2021. Systematic Review of Ethereum Smart Contract Security Vulnerabilities, Analysis Methods and Tools, Technische Universität Wien (Ed.).

[15] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. 2022. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. *Frontiers in Blockchain* 5 (2022). https://doi.org/10.3389/fbloc.2022.814977

[16] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart Contract: Attacks and Protections. *IEEE Access* 8 (2020), 24416–24427. https://doi.org/10.1109/ACCESS.2020.2970495

[17] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1361–1378. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[18] Xiangyan Tang, Ke Zhou, Jieren Cheng, Hui Li, and Yuming Yuan. 2021. The Vulnerabilities in Smart Contracts: A Survey. In *Advances in Artificial Intelligence and Security*, Xingming Sun, Xiaorui Zhang, Zhihua Xia, and Elisa Bertino (Eds.). Springer International Publishing, Cham, 177–190.

[19] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats. https://doi.org/10.48550/ARXIV.1811.06632

[20] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2021), 1133–1144. https://doi.org/10.1109/TNSE.2020.2968505