Karlsruhe Institute of Technology

Security Analysis Tools for Solidity Smart Contracts: A Comparison Based on Real-World Exploits.

Master's Thesis by

Michele Massetti

at the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Bernhard Beckert, Prof. Valentina Gatteschi
Advisor:	Jonas Schiff

15 April 2022 – xx MONTH 20XX

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, xx MONTH 20XX

. . . .
(Michele Massetti)

Abstract

A blockchain is essentially a digital ledger of transactions that is duplicated and distributed across the entire network of computer systems on the blockchain. Ethereum is considered a "smart contracts platform" which allows the development of smart contract applications. Those are self-verifying programs running on top of the blockchain.

Smart contracts are public, distributed and immutable, consequently, developers cannot design security by layer. The code must be examined for any potential weaknesses because they could lead to significant financial losses and harm public trust. The regulation of this field is not strict and smart contracts could hold millions of dollars. For these reasons, malicious users computed a great number of attacks, adopting different strategies

To guarantee security, numerous tools have been created, and a large amount of data about vulnerabilities and detection techniques is continually being produced. The security analysis tools implement the computer science security techniques, adapted for the blockchain context, such as fuzzing and symbolic execution.

This thesis is addressed to deepen the field of security in smart contract programming, written in Solidity: the most used and maintained programming language in this field. This work presents a collection of tools and real-world exploits

This work presents a collection of attacks and a collection of tools, selected after a literature research phase. Those are used for scanning real-world exploits that have occurred during the last two years (since 2020). The aim is to provide a comparison between those based on real cases. The comparison involves parameters such as the time of the installation requirements, the time of execution, the configuration of settings and the amount of discovered vulnerabilities. Furthermore, the tools are grouped based on their different characteristics, their typology, and even on their running mode.

Acknowledgement

Prova

Contents

List of Figures

List of Tables

1 Introduction

Blockchain represents one of the most popular trends in finance and computer science. It is a fully-distributed public ledger and a peer-to-peer platform. Cryptography is the base for securely hosting applications, transferring digital currency/messages and storing data. Bitcoin can be considered the “father” of this technology. **Bitcoin** depicted that in his paper, in the early 2009, it was effectively launched and the cryptocurrency Bitcoin was introduced. It works as a decentralised database.

Many blockchain systems have been born with new capabilities, which have allowed them to fit many different use cases. The first, which allowed developers to code on top of itself, was Ethereum. **Ethereum** published its whitepaper in 2014, and in 2015 it was deployed. The revolutionary aspect of Ethereum is the introduction of Smart Contract. These are programs running on blockchain systems and allow the developers to interact directly with this new technology. The development of innovative and prominent applications is a consequence of their development, such as NFT marketplaces, music royalty tracking, supply chain and logistics monitoring, voting mechanism, cross-border payments, decentralised finance and many others ([**BcUseCases**]).

The implementation of financial products enticed many investors during the last few years. The number of investments has been growing exponentially. According **statista**, the crypto market’s highest value reached around $3 trillion in 2021.

Considering Bitcoin, the cryptocurrency with the highest market capitalization, **CoinGeko** states its value around $38,553.70 and its market capitalization more than $700 billions.

Interest in such a valuable 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.

The topic that will be addressed in this thesis work is the comparison of security analysis tools for smart contracts, based on real-world exploits, so attacks that have happend during the recent years. It involves the understanding of smart contracts properties and the usage of different tools, providing insight regarding their behaviours in different contexts.

This thesis does not involve well-known benchmarks, with already studied vulnerabilities, but the tests are smart contracts involved in attacks. Our literature research faced off wide documentation of comparison of tools based on benchmarks and studied vulenrablities, but this work's ambition is to verify the effectiveness of the tools in real cases.

1.1 Research Goals

The main goal of this thesis can be summerized with the following research question:

How do state-of-the-art analysis tools for Ethereum/Solidity perform on real-world exploits?

This thesis involeves eight security analysis tools, which are chosen based on a literature research and on the type of analysis, trying to have a range of different typologies. Their analysis targets are smart contracts, involved in attacks, which have occured in the last two years (since 2020). One of the goals is the definition of the violated properties of those, understanding how they are computed by the attackers.

Furthermore, the comparison of the tools is based on a range of factors, these are some of the parameters used for providing a comparison and an evaluation of the tools.

- the performance;
- the completeness of the analysis;
- the facility of usage, involving the amount of code to be provided;
- the amount of found vulnerabilities;
- the report interpretability;
- the time for the configuration;

The research question deals with different topics, which can be expressed with the following sub questions:

1. How does a tool perform the analysis?
2. Which properties have been violated in the real-world exploits?
3. Which vulnerabilities are the tools able to detect?
4. In which context a tool perform better?

1.2 Releted Works

Nowadays multiple surveys and research work addressing smart contracts analysis have been published. The ones, we are interested in, deal with the review of vulnerabilities, description and comparison of tools and definition of new techniques for scanning those.

The selection of tools was anticipated by research work. We picked those starting from surveys and papers, regarding comparison of multiple of tools, such as **Survey1**, **Survey2**, **Survey3**, **Survey4**, **thesis**. These give a general overview and provide a comparison based on different aspects: type of installation, running mode or type of analysis. A taxonomy is provided as well. For having a deeper knowledge of every single tool, we considered their papers and documentation.

In this thesis, we involved automated tools (**Slither**, **Mythril**) and the ones which provide the possibility to run custom analysis. The first ones have as targets vulnerabilities such as reentrancy, overflow/underflow, and gas exceptions; but they do not provide functional correctness guarantees. On the other hand, the second group try to solve these issues by providing more possibilities for modelling the analyses. We involved tools adopting formal verification (**CertoraDocumentation**, **SolcVerify**, **CelestialPaper**) and fuzzing (**Echidna**).

The cited works provide a comparison based on well-known benchmarks, defined vulnerabilities or just on the specifications. This work provides a comparison between the considered tools as well, but we tried to move a step forward. Rather than considering defined vulnerabilities, we consider real-world exploits, which have happened in the last couple of years.

The considered tools are installed and run on real-world attacks; these are chosen based on their effectiveness and the damage, in terms of drawn liquidity.

2 Preliminaries

The following sections are addressed to provide an introduction to the topics involved in our work: Ethereum, smart contracts and security techniques. These are necessary for understanding specific security vulnerabilities and the results obtained from our experiment. We recommend reading this chapter for those who are not experts on these topics, otherwise, the reader can directly jump at **??**.

2.1 Blockchain

Blockchain technology is a specific subset of the broader distributed ledger technology (DLT) universe that uses a particular data structure consisting of a chain of hash-linked blocks of data.

DLT has the role of an umbrella term to encompass multi-party systems that operate in an environment with no central authority, despite parties who may be unreliable or malicious. A distributed ledger is described, by **DLT**, as a type of distributed database that assumes the preseence of nodes with malicious intentions. A distributed leger comprises a ladeger's multiple replications in which data can only be appended or read. The ledger, by definition, store the records of transaction, which are final definitve and immutable. A transaction is the smallest unit of a work process, which is one or more sequences of actions required to produce an outcome that complies with governing rules ([**Ledger**]).

Blockchain contains a complete history of all data transfers and transactions that have ever taken place since it was first created. In other terms, it keeps a collection of records called blocks. Any updates or validations on the network reflect in all copies simultaneously. This guarantees the fidelity and security of data records and generates trust in the system, without the need for a centralized trusted third party. Blockchain networks rely on consensus algorithms to reach agreement among various distributed peers.

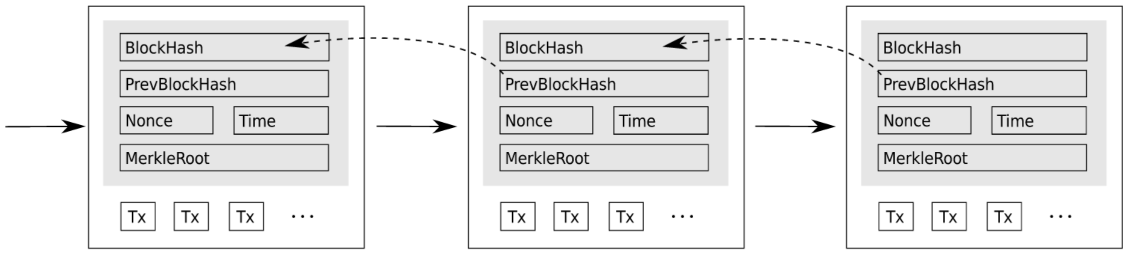

Figure 2.1: Simplified blockchain (image from [**BlockChain1**]).

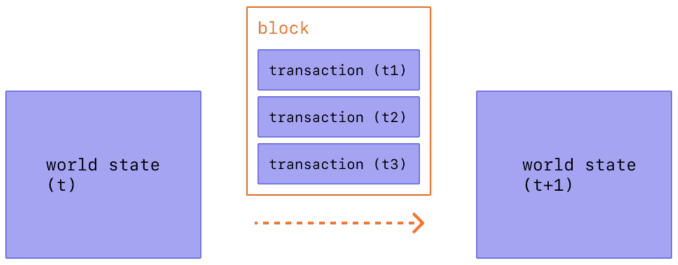

Figure 2.2: Change state overview(image from [**EthDocs**]).

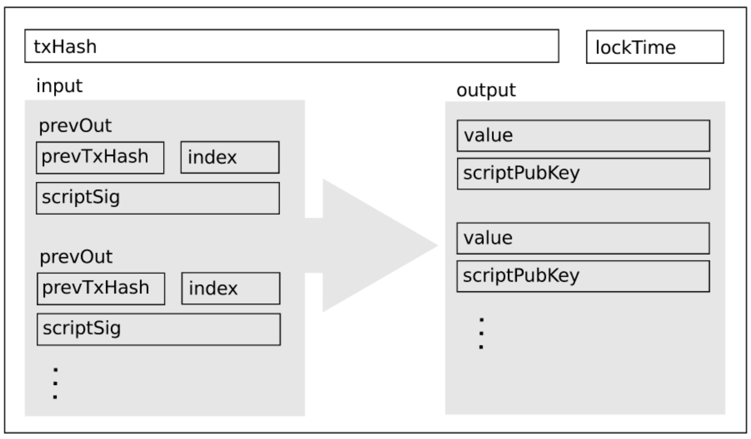

Figure 2.3: Trasnactions scheme (image from [**BlockChain1**]).

The term "blockchain" is formed by "block", which refers to state and data being stored in consecutive groups, and "chain", which describes how each block contains a cryptographic reference to its parent.

Blocks are collections of transactions that have the hash of the block before them in the chain. Because hashes are cryptographically formed from the block data, this connects blocks. Because any alteration to any historical block would render all subsequent hashes incorrect and cause everyone using the blockchain to become aware of it, this prevents fraud. These concepts are shown, in a simplified version, by **??**. Each block contains a list of transactions and keeps the reference to the previous one.

Transactions (**??**) are cryptographically signed instructions from accounts, which are entities, having the possibility to get a balance of native token and interact with the blockchain. Accounts can be user-controlled or deployed as smart contracts. An account, when it is enrolled in the system, receive a couple of keys: private and public one. An asymmetric algorithm is implemented. A user is identified by the public key and can make a transaction by signing it with the private key.

An account will initiate a transaction to update the state of the network. These are collected and stored in blocks, which then are broadcasted in the network. When this happens, a change in the state of the chain is applied (**??**). This procedure, firstly the storing of transactions in blocks, and then those are the ones broadcasted on the net, is due to the fact the state of the blockchain is kept in the blocks, and not in the transactions.

The entities, the computers, taking part in the network, are called "nodes". Each of those must agree upon each new block and the chain as a whole. Nodes guarantee that everyone using the blockchain has access to the same data.

A consensus mechanism ([**Consensus**]) is required for blockchains to implement this distributed agreement. It is a process in computer science used to achieve agreement on a single data value among distributed processes or systems. A consensus algorithm secures the network and prevents unauthorized users from validating bad transactions. The mechanism also enables agreement on the network even when no single node is in charge. Proof of work (PoW) and proof of stake (PoS) are the two mostly used ones.

Following the survey of **BlockChain1**, we can depict four main characteristics of this technology:

- decentralization, due to the absence of third party control;
- persistency, because of the impossibility to delete or modify any transactions once they are stored in the ledger;
- anonymity, regarding the cryptographic aspects, a user is identified just by the asymmetric keys, public and private ones;
- auditability, that means that each pear can verify by itself the validity of block verification.

2.1.1 Smart Contracts

Smart Contract is a concept which is directly associated with blockchain, and specifically with Ethereum. However, its concept was defined even before the first blockchain, in the 1994 by **SmartContractDef**. Its goal is to encode and enforce the entire logic that governs an interaction between two (or more) mutually distrustful parties. By design it is autonomous, so it can be seen like a piece of software, made by codes. Its strenght is the possibility to play the role of a trusted intermediary between distrusting parties without being an actual party itself. Blockchains are the first platform that allow the actual realization of Smart Contracts without a trusted third party.

The main limitation of those deals with their their inability to interact with the "external world", only the accounts can alterate the state of the systsem. Since they have to be deterministic so that all nodes arrive at the same result for each computation. Basically, those cannot get information about "real-world" events on their own. The main reason of that involves that relying on external information could undermine consensus, because external calls return different things at different times. For example a smart contract is not allowed to send HTTP requests or interact with any APIs.

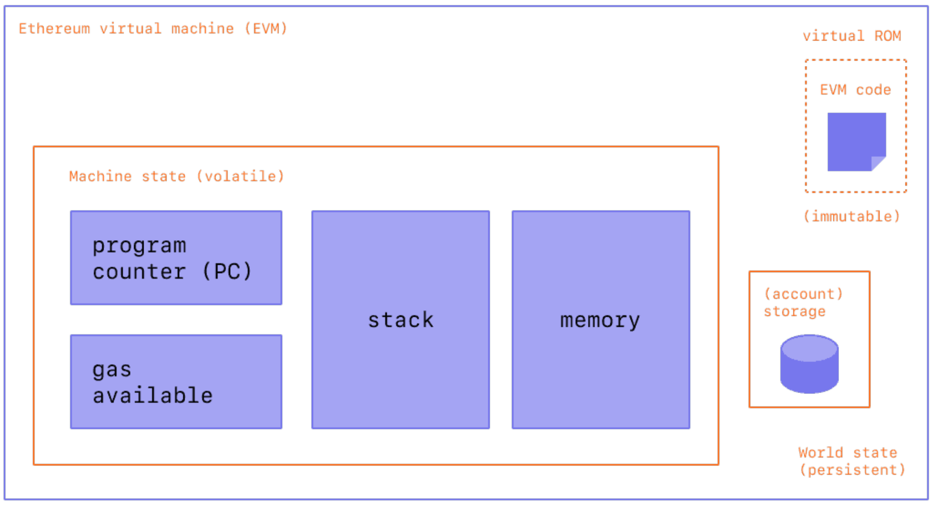

Figure 2.4: Diagram of EVM (image from [**EthDocs**])

More limitations are introduced based on the blockchain, for example Bitcoin does not allow an arbitrary behaviour and the size of those is reduced, considering other systems, as Ethereum, which gives much more freedom of operability.

2.2 Ethereum

Ethereum is a blockchain that was developed by **Ethereum**, presented in the 2014. In the Ethereum universe, a single, canonical computer called "Ethereum Virtual Machine" (EVM) represents the abstraction of the ledger, and everyone on the Ethereum network agrees on its current state. Every node, or participant in the network, maintains a copy of this computer's state. **??** illustrates a schematic representation of the EVM, underlying the common aspects with a computer, as the stack, memory, the storace, the EVM code (the smart contracts) and so on.

The blockchain, which is saved and approved by all nodes, keeps track of all transactions and the current state of the EVM. Once a transaction is confirmed as genuine and put to the blockchain, cryptographic methods ensure that it cannot be changed later.

Its consensus mechanism is defined as "proof of work" (PoW). It provides the rules for defining the node, which gets the responsability, and consequent rewards, of collecting the transaction, generating a block and broadcast it. Those nodes are called "miners", due to the mining, which is the process of creating a block of transactions to be added to the Ethereum blockchain.

The nonce for a block must be discovered by miners through a rigorous process of trial and error under the proof-of-work protocol. A block can only be added to the chain if its nonce is correctly verified by the other peers. The miners take part in this sort of race and the one who succedes to find the nonce, verified by the other peers, has the right to publish the block and get the rewards.

The rewards are paid in the native token of the Ethereum: ether (ETH).

2.2.1 Ethereum Smart Contracts: Solidity

EthDocs describes its technology as a "Smart Contract Platform". Those are described as programs that run on the Ethereum blockchain. Their EVM-bytecode and data are stored on the Ethereum blockchain, reachable at a specific address. Since they are Ethereum account, they have a balance and can be the target of transactions.

A smart contract can be created by anyone and released across the network. Technically, deploying a smart contract counts as a transaction, so the deployment has a cost, called "gas fees". Gas prices for contract deployment, however, are significantly higher.

Smart contracts represent self-autonomous and self-verifying agents stored in the blockchain. They are composed of fields and functions ([**SmartContracts**]). After their deployment in the blockchain, they have their unique address which the users/clients can use to interact with. An "external account" is controlled by public-private keys used by humans, on the other hand, a "contract account" is referred to those. **??** gives an overview of the process of smart contracts running. The code that is stored in the blockchain after deployment is a low-level stack-based bytecode (EVM bytecode) representative of the high-level programming language (e.g. Solidity) in which the smart contracts are initially written.

Smart contracts' behaviour is completely predictable and its code can be inspected by every node in the network because the bytecode is publicly available from the blockchain. **EthDocs** explains that EVM executes as a stack machine, a virtual machine in which the primary interaction is moving short-lived temporary values to and from a push down stack ([**stack**]). Compiled smart contract bytecode executes as a number of EVM opcodes, which perform standard stack operations like XOR, AND, ADD, SUB, etc.

During execution, the EVM maintains a transient memory (as a word-addressed byte array), which does not persist between transactions. All the changes are applied to the system when the transaction is published, if it reverts, all the changes are lost.

EthDocs states that Solidity is the most used and mantained high level program language for smart contracts. It is similar to the JavaScript programming language which after deploying is compiled into EVM bytecode.

Its main features can be summarized as follow:

- object-oriented, high-level language for implementing smart contracts;
- statically typed (the type of a variable is known at compile time);
- curly-bracket language that has been most profoundly influenced by C++;
- the support of inheritance, libraries and complex user-defined types.

A particular type of function in Solidity is the "fallback", which is used even by attackers for malicious aims. Its characteristics are the following:

1. triggered when a non-existent function is called on the contract or when it receives a number of native tokens;

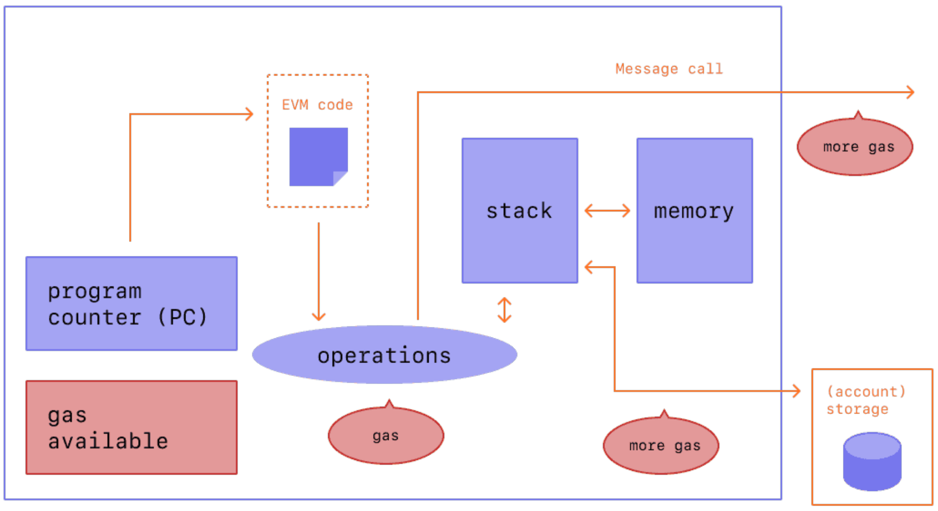

Figure 2.5: Diagram of EVM runnnig smart contracts (image from [**EthDocs**])

2. it is marked as external (not callable by functions of the same contract);
3. it has no name and arguments;
4. the return value is empty;
5. a contract can have just one of it;
6. if not marked payable, it will throw an exception if the contract receives plain ether without data.

The first property is fundamental in the case of "reentrancy" attacks. It occurs when the workflow of a smart contract is modified by an external call. Due to the fallback, the malicious external contract can modify the order of the execution of the target. An effective example occurs when a contract modifies the variable after the external call, so the malicious one can call back the same function (like recursion), but the variables are not updated.

This thesis considers attacks that occurred even in other blockchains since those are defined as "EVM-compatible". That means the infrastructure of the systems has certain parts of the Ethereum network. This approach turned out to be beneficial for the developers of other blockchains, both in terms of time, knowledge transfer, and maybe most importantly interoperability. Having a code execution environment similar to Ethereum's Virtual Machines, Solidity is the programming language, consequently, same contracts can run on multiple blockchains.

2.3 Application Domains

Smart contracts technology gave the possibility to implement new financial instruments, based on their decentralised characteristic.

Since the cryptographic aspects of the blockchain, it allows giving uniqueness to digital data thanks to the NFTs.

Decentralised Finance Multiple investors have been attracted by new possibilities offered by "Decentralised Finance" (DeFi). An article on the European Central Bank website ([**DeFiDef**]) deepens this topic. The main strength of these financial instruments is the possibility to cut out traditional centralised intermediaries and rely on automated protocols instead. In this case, the investors are part of a peer-to-peer network where assets represented in the network can be transferred automatically, thanks to smart contracts. In most cases, the DeFi applications do not provide new financial products or services but mimic within the crypto-asset ecosystem those provided by the traditional financial system.

Since there are not any legal regulations and many investments have been involved in DeFi, multiple malicious hackers have exploited this lack in their favour.

One of the targets of the attackers is a particular type of decentralized exchange (DEX) named autonomous market maker (AMM). **AMMDef** presents it as a DEX based on a mathematical formula. Its peculiarity is how it estimates the asset price: according to a pricing algorithm. It is considered autonomous and decentralised because the user does not need to have a counterparty (another trader) on the other side to make a trade. On the other hand, users directly interact with a smart contract that “makes” the market.

Since there are no counterparties, the market is generated by providing liquidity in the smart contract by the users forming liquidity pools (LPs). First of all, liquidity, clarified by **LiquidityDef**, is the measure of how easily you can convert an asset into cash or another asset. In other words, it depicts how the conversion of an asset into another one affects the price of the asset. Liquidity is positively estimated when the asset’s price is not easily affected by the transactions of the asset and the asset is easily sold and bought.

A liquidity pool can be seen as a big pile of funds that traders can trade against. In return for providing liquidity to the protocol, LPs earn fees from the trades that happen in their pool.

NFT Non-fungible tokens (NFTs) are cryptographic assets on a blockchain with unique identification codes and metadata that distinguish them from each other. Unlike cryptocurrencies, they cannot be traded or exchanged at equivalency. This differs from fungible tokens like cryptocurrencies, which are identical to each other and, therefore, can serve as a medium for commercial transactions.

2.4 Security Analysis

Smart contracts are by design public and available by every account. In most cases, the code is published, otherwise, the bytecode is always available. This aspect of the blockchain represents a drawback in sort of security because establishing security by layers scheme is impossible. The developers cannot configure any sort of VPN (Virtual Private Network), Firewall or IDS (intrusion detection system). The attackers can directly access and interact with the smart contracts, consequently, the discovery of vulnerabilities turns into a challenge.

Smart contracts cannot be changed after deployment. For this reason, discovering security issues is an even more severe challenge, and it is an argument for static analysis/formal methods that can be done before deployment.

To fulfil this goal, multiple techniques have been adopted.

Fuzzing **OWASPFuzz** gives us an overview of this Black Box software testing technique. The goal is to find bugs, using random/semi-random data injection in an automated way.

A fuzzer, which is the program implementing this, injects automatically the data, generated starting from a seed. The inputs depend on the generators. These can involve just a seed, or even static fuzzing vectors (known-to-be-dangerous values). The number of possible solutions is infinite, consequently, fuzzers fix a threshold of cases. One of the strengths of the systematic/random approach is to find bugs that would have often been missed by human eyes.

Formal Verification Formal verification has the aim to prove or disprove the correctness of a system by checking its mathematical model against a certain formal specification. A specification is a set of properties, which involve the behaviours of the software. Models and specifications can regard different levels of abstraction via various types of formalisms.

[**FormalVerificationDef**] explains that the absence of design flaws is not possible to prove, but the formal verification looks for the complete space of possible design behaviours which can be used to find and eliminate errors.

However, a drawback is that a timely solution is not always guaranteed in certain situations. It should have full coverage, but in real cases, the tools have a certain threshold considering time or an amount of computational effort.

Symbolic Execution **SymbolicExecution** provides a definition of this technique and its possible variations. Symbolic execution can be considered as a subgroup of Formal Verification. It is a program analysis technique to test whether certain properties can be violated by a piece of software.

The strength of symbolic execution is the simultaneous exploration of multiple paths that a program could take under different inputs. This paves the road to sound analyses that can yield strong guarantees on the checked property. The key idea is to allow a program to take on symbolic, rather than concrete, input values.

Symbolic execution is less general than abstract interpretation because it does not explore all paths through the program. However, symbolic execution can avoid giving false warnings; any error found by symbolic execution represents a real and feasible path through the program, and a test case can illustrate that error.

3 Exploits

In this part of thesis, we introduce the objectives of the analyses: the real world exploits. **??** illustrates the attacks selected for this work, providing a brief description of those.

This thesis involves real-world exploits over the last two years since 2020. Our choice is based on the typology of the attack, how it was computed, and the target of those. Our work involves the security of smart contracts, written in solidity. For having a wider overview of the topic, we decided not just to provide smart contracts deployed on the Ethereum chain, but even the ones deployed on other chains, specifically Avalanche and Binance Smart Chain. Those are still EVM-compatible blockchains.

The attacks have as target DeFi protocols. These involve the usage of smart contracts for the implementation of financial products, involving the management and distribution of valuable assets. Because of the impossibility of security by layer, the implementation of a financial system with smart contracts turns into a challenge.

The selected exploits deal with projects, involving a great number of investments. The targets of attacks are not just autonomous money makers (AMM), but they can involve the implementation of a token or NFTs. **NFTValue** estimates the value of NFT market around $100 billion. Nowadays, the word NFT is one of the most researched ones on Google and the other search engine. NFT's marketplaces manage the transaction behind these valuable markets.

The selection of those involves not just the consequent damage of the exploit, but the target and the vulnerability.

This chapter contains a subsection per each exploits. The general structure of those involves the following three main points:

Table 3.1: Exploits

Exploits	Detail
Aku	Funds stuck in the contract
Cover	Wrong update of data
BZX	Wrong implementation of token
Spartan	Bad estimation of rewards in a pool
Uranium	Vulnerable deposit withdraw process in a pool
XSURGE	Reentrancy in a token
BurgerSwap	Reentrancy in an AMM
DirtyDogs	Reentrancy in NFTs market

1. the explanation of the protocol;
2. the exploit;
3. the security properties of the smart contracts.

The first part gives a general overview of the smart contract, defining its goals and main characteristics. After that, we present the exploit, illustrating the vulnerable part of the source code and the adopted strategies by the attackers. In conclusion, their properties are depicted. These are elaborated per each tool for defining the specifications. Per each tool, the specifications are written in terms of specifications, needed for the analysis.

3.1 $34 Million stacks NFT Project Aku Dreams Smart Contract

Akutars is an Ethereum-based NFT project developed by Aku Dreams. **Aku** reports Akutarts locked up $34 million due to the faulty code of the smart contract. The launch contained 15,000 NFTs and was based on the Dutch auction. This strategy involves a descending price auction where an item begins at a set maximum price. The price is gradually lowered over a fixed time until a bid is placed that guarantees the bidder to purchase the item at the current price. The smart contract had the role to keep the bids, paying in ETH, and then refunding the users who could not succeed in the auction.

An attacker blocked refunds of all the bidders who could not get the NFTs.

3.1.1 The exploit

The first part of the exploit involved the function `processRefunds` **??**.

This has the aim to refund the bid of the user who took part in the auction and did not get the NFTs.

The for loop is the vulnerable part of the code. It loops on all over the users to be refunded, estimating the number of tokens to send. Then, the amount is sent with the function call, which returns a boolean, based on the correct execution of the operation.

The `require` statement represents the vulnerability. If one of the accounts could not receive the refund, the function would always revert. Since looping all over the users is a sequential operation, if the transaction just reverted when it reaches an item, it would never reach all the following items.

Therefore, a malicious user just implemented a smart contract which took part in the auction and reverted any time it received tokens.

```
  function processRefunds() external {
    ...
    for (uint256 i=_refundProgress; gasUsed < 5000000 && i < _bidIndex; i++) {
        bids memory bidData = allBids[i];
        if (bidData.finalProcess == 0) {
          ...
          if (refund > 0) {
              (bool sent, ) = bidData.bidder.call{value: refund}("");
              require(sent, "Failed to refund bidder");
```

```
            }
          }
          ...
          _refundProgress++;
      }
      refundProgress = _refundProgress;
    }
```

Listing 3.1: Function for refunding the users.

This project contains another bug, which could not allow the developer team to withdraw the project funds. The function `claimProjectFunds` (??) is callable only by the owner of the contract due to the modifier onlyOwner, refunds the developers just when all the users are considered refunded.

The statement of `require` deals with varibles which are badly estimated It compares the variable `refundProgress`, which takes track of the refund progress, and `totalBids`.

```
    function claimProjectFunds() external onlyOwner {
        require(refundProgress >= totalBids, "Refunds not yet processed");

        (bool sent, ) = project.call{value: address(this).balance}("");
        require(sent, "Failed to withdraw");
    }
```

Listing 3.2: Function for claiming the funds for the developers.

The function `_bid`, called by a user for placing an arbitrary amount of bids, increases the variable `totalBids` based on the amount of placed bids.

?? shows this process. The variable `refundProgress` is increased every time a user is refunded. Consequently, if a user bought more than one bid, the number of refunded users would never be greater or equal to the number of placed bids.

```
    function _bid(uint8 amount, uint256 value) internal {
        ...
        uint256 myBidIndex = personalBids[msg.sender];
        ...
        if (myBidIndex > 0) {
            myBids = allBids[myBidIndex];
            refund = myBids.bidsPlaced * (myBids.price - price);
        }
        ...
        uint256 _totalBids = totalBids + amount;

        if (myBidIndex > 0) {
            allBids[myBidIndex] = myBids;
        } else {
            myBids.bidder = msg.sender;
            personalBids[msg.sender] = bidIndex;
            allBids[bidIndex] = myBids;
            bidIndex++; //bidIndex updated just when a new user computes a bid
        }
        totalBids = _totalBids; //totalBids depends on the amound of bids
        ...
```

```
}
```

Listing 3.3: Function for users'bid

3.1.2 Properties

The smart contract involves 2 main problems: the refunding of the users who placed the bids and the claim of the developers' rewards.

The first property deals with the function `processRefunds`. It reverts every time because a malicious wallet, which cannot receive any tokens triggering the `require`. The propety involves the estimation of the refunding users. This can be expressed as the sum of refunded wallets equal to the counter of loops on the map containing all the users. This is expressed by the following postcondition:

$$\_refundProgress == \_counter_of_loops \tag{3.1}$$

If the property always succedes, that means the function reverts every time in case of an error, without considering an error in sending the transaction.

The other property regards the function `claimProjectFunds`. Our focus is on the comparison between the counter of the refunded users and the total amount of bids. The property is the comparison between the `processRefunds` and `totalBids`. The aim is to check if it feasible that the number of refund users is less than number of placed bids. The opposite of the boolean statement of the `require` of the function. If the property is always realised, that means that there is no feasibility to execute the function.

The precondition is :

$$refundProgress < totalBids \tag{3.2}$$

3.2 Cover Protocol:Infinite Minting Exploit Nets Attacker $4.4M

In December 2020, an exploit was abused on Cover Protocol's shield mining contract. The article shows the attackers stole from the project around $ 4 million. Its bug had the result to mint more rewards to the miner.

CoverProtocol interview gives an overview of the project. It is deployed on the Ethereum blockchain and is a peer-to-peer coverage marketplace that utilizes ERC-20 fungible tokens to allow permissionless and non-KYC coverage. It can be described as a coverage provider. The attack affected the rewards contract, consequently, the token's one even. The exploit can be classified under the name of "infinite mint".

3.2.1 The exlpoit

The developers' team reported [**CoverProtocolPostMortem**] the technical analysis of the exploit the day after. The core protocol was not affected, but the minting contract and the $COVER token became unusable. Firstly, the attackers created a new balancer liquidity pool for the target contract. The next step was to deposit tokens in it and execute the exploit, withdrawing funds from the contract thanks to a miscalculation of the rewards. The bug relies on the misuse of two keywords in solidity: storage and memory.

Memory This keyword within Solidity allocates memory for a specific variable. In this instance, that variable is scoped to a specific function. The memory is cleared once the function has executed.

Storage On the other hand this keyword within Solidity allows variables to act as a pointer into the storage of data in mappings or data structures. Storage data is persistent between function calls and transactions.

The previous has a similar behave to the Random Access Memory (RAM) on a computing device, the latter stores into the persistent memory.

The vulnerable function is the deposit one.

```
    function deposit(address _lpToken, uint256 _amount) external override {
        ...
        Pool memory pool = pools[_lpToken];
        updatePool(_lpToken);

        Miner storage miner = miners[_lpToken][msg.sender];
        BonusToken memory bonusToken = bonusTokens[_lpToken];
        _claimCoverRewards(pool, miner);
        _claimBonus(bonusToken, miner);
        // variable pool is not updated
        miner.rewardWriteoff = miner.amount.mul(pool.accRewardsPerToken).div(CAL_MULTIPLIER)
    ;
        ...
  }
```

Listing 3.4: Deposit function.

?? shows the state of the pool is stored in a variable with the keyword memory. The function update updates the state of the pool. However, the variable pool, existing within the function, remains identical.

The function estimates the reward per token updating the value of miner.rewardWriteoff, but it uses the wrong value of the parameter of pool.accRewardsPerToken.

Following the vulnerability, anyone can obtain an insane amount of minted tokens when they execute the claimRewards(address _lpToken) function. This function, which is used to grab their rewards, ends up calling _claimCoverRewards(Pool memory pool, Miner memory miner) which references the miner.rewardWriteoff. As that variable is much smaller than the actual pool.accRewardsPerToken, the contract results in minting an abundance of tokens.

3.2.2 Properties

The heart of the problem is the wrong management of the keywords storage and memory.

The consequence of this error is a miscalculation of the reward of the miner. It is not estimated considering the correct parameters of the pool.

The property is a posticondition of the function `deposit`. It compares the variable `miner.rewardWriteoff` with its recomputed value with the updated parameters, right after the execution of the function.

$$miner.rewardWriteoff == miner.amount.mul(pool.accRewardsPerToken) \\ .div(CAL\_MULTIPLIER) \tag{3.3}$$

The variable is not computed with the updated value of the pool: the reason for the exploits.

3.3 DeFi platform bZX: $8M hack from one misplaced line of code

bZxProtocol explains how this protocol works. Anyone can use bZx to create apps that allow lenders, borrowers, and traders to interact with Ethereum based decentralised finance protocol.

Protocols can be developed by bZx. It is a DeFi platform for tokenized lending and margin trading. iTokens (margin loans) represent the earn holders' interest on borrowed funds allowing margin positions to be composable.

It suffered a couple of attacks in February 2020. The developers explained the attackers could drain different currencies,219,199.66 LINK, 4,502.70 Ether (ETH), 1,756,351.27 Tether (USDT), 1,412,048.48 USD Coin (USDC) and 667,988.62 Dai (DAI): a total of $8 million in value.

3.3.1 The exploit

The object of the attack was the contract which implements the logic behind the protocol. Every ERC20 token has a `transferFrom` function, which has the aim to transfer the tokens. Calling this function allowed the attacker to create and transfer an iToken to hitself: his balance could be artificially increased. The duplicated tokens were then redeemed for their underlying collateral, with the hackers now "owning" a much higher percentage of the pool, so the attacker could withdraw the tokens.

?? shows the vulnerable function. The attacker called that with the same address for `_from` and `_to` parameters.

The estimation of the balance is wrongly computed. Additional variables are used for computed those. Firstly, the balnce of the sender is computed. The balance of the receiver is computed as the sent amount and its value stored at the beginning in the additional variable. Therefore, if the addresses are the same, the balance increases since it is computed as the sum of the sent amount and its previous value.

```
...
function _internalTransferFrom (address _from, address _to, uint256 _value, uint256
_allowanceAmount)
    internal
    returns (bool)
{
        //The heart of the vulnerability
        uint256 _balancesFrom = balances[_from];
        uint256 _balancesTo = balances[_to];

        require(_to != address(0), "15");

        uint256 _balancesFromNew = _balancesFrom
            .sub(_value, "16");
        balances[_from] = _balancesFromNew;

        uint256 _balancesToNew = _balancesTo
            .add(_value);
        balances[_to] = _balancesToNew;
        ...
    }
```

Listing 3.5: Vulnerable function in LoanTokenLogicStandard contract.

The developers corrected the bug in few days. It was enough switching some lines of code, in order to avoid the operations of sum and subtraction operate on the same balance. The code **??** presents some differences. The operations regarding the receiver's balance are computed, then those which deal with the sender's one.

```
    function _internalTransferFrom(address _from, address _to, uint256 _value, uint256
    _allowanceAmount)
        internal
        returns (bool)
    {
        ...
        require(_to != address(0), "15");
        uint256 _balancesFrom = balances[_from];
        uint256 _balancesFromNew = _balancesFrom
            .sub(_value, "16");
        balances[_from] = _balancesFromNew;
        uint256 _balancesTo = balances[_to];
        uint256 _balancesToNew = _balancesTo
            .add(_value);
        balances[_to] = _balancesToNew;

        return true;
    }
```

Listing 3.6: Corrected bug in LoanTokenLogicStandard contract.

3.3.2 Properties

The function `internalTransfer` is the one which contains the bug abused by the attackers.

The preperty is a postcondition. It states the correct estimation of the balances of the addresses involved in the operation. The balance of the sender should decrease of the sent amount and the one of the receiver should increase of the sent amount.

$$\begin{aligned} old\_value\_of(balances[\_from]) >= balances[\_from] \\ \&\&old\_value\_of(balances[\_to]) <= balances[\_to] \end{aligned} \tag{3.4}$$

3.4 A flash loan used for amplify a bug: $30M drained from Spartan protocol

Spartan Protocol is a DeFi protocol for synthetic assets running on BinanceSmartChain. It inherits many capabilities of UniswapV2 protocol, adapting the code for new use cases and implementing different strategies. The fee mechanism is modified to incentivize liquidity providers when liquidity is scarce. Consequently, users trading larger volumes are charged more fees. Similar to UniswapV2, pairs WBNB and SPARTA tokens are open for users to add/remove liquidity.

This protocol was the target of an exploit at the end of May 2021. The presence of a bug inside the code, plus the amplification due to a flash loan, allowed the attacker to drain the liquidity.

The articles **FlashCoin** and **FlashCloud** give a defintion flash loan:

Definition 1. *A flashloan is a relatively new type of uncollateralized lending that has become popular across several decentralized finance (DeFi) protocols based on the Ethereum network. When it has been issued, the smart contract certifies that the borrower pays back the loan before the transaction ends. If this condition is not fulfilled, the transaction reverts, consequently, the amount of the loan is given back.*

The role of the flash loan is to amplify the effectivenss of the attack.

3.4.1 The exploit

The exploit gas as target the bad implemetation of the strategy for the management of the liquidity in the pool. The mistake of the developers was not to consider the updated value of underlying assets. Those are stored into the variables `baseAmount` and `tokenAmount` and estimated with `iBEP20(token).balanceOf(pool)` and `iBEP20(base).totalSupply()`.

The bug in code lies in the `calcLiquidityShare` function, called in `RemoveLiquidity`.

```

    function calcLiquidityShare(uint units, address token, address pool, address member)
    public view returns (uint share){
        // share = amount * part/total
        // address pool = getPool(token);
        uint amount = iBEP20(token).balanceOf(pool);
        uint totalSupply = iBEP20(pool).totalSupply();
        return(amount.mul(units)).div(totalSupply);
    }
```

Listing 3.7: calcLiquidityShare function

It should get the balance of the underlying asset in the pool (??). The amount that which should be transferred out is calculated based on the total LP tokens supplied and the number of LP tokens to burn (units). The function does not consider who transfers assets into the pool. The value of underlying assets can be manipulated and increased by an exploit. The real values are different from the ones contained in the variables `baseAmount` and `tokenAmount`. The `removeLiquidity` function calls `calcLiquidityShare` one on TOKEN and BASE(**??**). It fails to synchronize the balances of the underlying assets and the variables which store the amount of the assets.

```
// Remove Liquidity for a member
function removeLiquidityForMember(address member) public returns (uint outputBase, uint
outputToken) {
    uint units = balanceOf(member);
    outputBase = iUTILS(_DAO().UTILS()).calcLiquidityShare(units, BASE, address(this),
member);
    outputToken = iUTILS(_DAO().UTILS()).calcLiquidityShare(units, TOKEN, address(this),
 member);
    ...
    return (outputBase, outputToken);
}
```

Listing 3.8: Function for Removing Liquidity

As a consequence, the `_decrementPoolBalance`, updates the wrong value of the variables storing the assets. It does not get the update-to-date balances of BASE and TOKEN. Instead, it only decrements the reserved amounts. The attacker followed these steps for draining the liquidity:

1. Add liquidity and get LP tokens back.
2. Transfer some assets into the Pool contract to amplify the number of underlying assets of the LP tokens collected in step 1.
3. Remove liquidity and get more assets than what you added in Step 1.
4. Add the assets transferred into the Pool contract as liquidity and remove them immediately.

```
function _decrementPoolBalances(uint _baseAmount, uint _tokenAmount) internal  {

    uint _removedBase = iUTILS(_DAO().UTILS()).calcShare(_baseAmount, baseAmount,
baseAmountPooled);
    uint _removedToken = iUTILS(_DAO().UTILS()).calcShare(_tokenAmount, tokenAmount,
tokenAmountPooled);
    baseAmountPooled = baseAmountPooled.sub(_removedBase);
    tokenAmountPooled = tokenAmountPooled.sub(_removedToken);
    //Wrong Estimation of these 2 variables
    baseAmount = baseAmount.sub(_baseAmount);
    tokenAmount = tokenAmount.sub(_tokenAmount);
}
```

Listing 3.9: Function which updates decrements the assets in the pool.

A solution for this bug is shown in **??**. It updates the variables of assets before it is estimating the the amount to drain.

```
function calcLiquidityShareSynch(uint units, address token, address pool, address member
) public view returns (uint share){
    // synchronize the variable
    iPOOL(pool).sync();
```

```
        uint amount = iBEP20(token).balanceOf(pool);
        uint totalSupply = iBEP20(pool).totalSupply();
        return(amount.mul(units)).div(totalSupply);
    }

    function sync() public {
        baseAmount = iBEP20(BASE).balanceOf(address(this));
        tokenAmount = iBEP20(TOKEN).balanceOf(address(this));
    }
```

Listing 3.10: Possible corrcet calcLiquidityShare.

3.4.2 Properties

The issue correlated to the logic of the program involves the removing liquidity process. The attacker could amplify the number of tokens to remove.

The function `calcLiquidityShare` is an internal function which estimates the total of underlying assets to send to the user. It is called by the `removeLiquidityForMember`; it fails to synchronize the balance of the underlying assets into the reserved assets.

The property is a postcondition of the function `calcLiquidityShare`. It checks that the underlying assets variables correctly keep track of the current value.

$$\begin{aligned} iBEP20(token).balanceOf(pool) &== baseAmount \\ ||iBEP20(token).balanceOf(pool) &== tokenAmount \end{aligned} \tag{3.5}$$

3.5 Uranium Finance: $1.3M of rewards drawn

Uranium Finance is a Automated Marker Maker (AMM) runnning on the BinanceSmartChain. The article presented by **UraniumPM**, deals wiht the exploit which occured on the 8th April 2021. The attacker could grab the contents of the RADS pool and all of the RADS/sRADS rewards and sell them for $1.3M worth of BUSD and BNB.

The team of developer could identify the exploiter, because some transaction of the attacker wallet, could be correleted with a Binance wallet. The criminal got in touch with the developers. After some negotiation, the exploiter refund the team of $1M in ETH.

3.5.1 The exploit

UraniumTech gets more in deep into the technical details involved in this exploit. Its target was the part of the protocol regarding the rewarding of the user. The list of transactions involving the malicious wallet shows the attacker could draw a huge amount of rewards by calling 3 functions multiple times:

1. `deposit(_pid, _amount);`
2. `emergencyWithdraw(_pid);`
3. `withdraw(_pid, _amount).`

Deposit The two most relevant variables to the exploit are user.amountWithBonus and user.rewardDebt, for the attack purpose, they need to be greater than 0. Therefor this function is called with with the texttt_amount input argument larger than “0”.

The `user.amountWithBonus` increases by adding the texttt_bonusAmount. The `user.rewardDebt` is calculated by the end of the function, ??.

```
function deposit(uint256 _pid, uint256 _amount) external validatePool(_pid) {
    ...
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(_user), address(this), _amount);
        if (address(pool.lpToken) == address(rads)) {
            uint256 transferTax = _amount.mul(2).div(100);
            _amount = _amount.sub(transferTax);
        }
        //The bonus increase
        if (pool.depositFeeBP > 0) {
            ...
            uint256 _bonusAmount = _amount.sub(depositFee).mul(userBonus(_pid, _user).
add(10000)).div(10000);
            user.amountWithBonus = user.amountWithBonus.add(_bonusAmount);
            pool.lpSupply = pool.lpSupply.add(_bonusAmount);
        } else {
            user.amount = user.amount.add(_amount);
            uint256 _bonusAmount = _amount.mul(userBonus(_pid, _user).add(10000)).div
(10000);
            user.amountWithBonus = user.amountWithBonus.add(_bonusAmount);
            pool.lpSupply = pool.lpSupply.add(_bonusAmount);
        }
    }
    user.rewardDebt = user.amountWithBonus.mul(pool.accRadsPerShare).div(1e12);
    emit Deposit(_user, _pid, _amount);
}

// Withdraw LP tokens from MasterUranium.
```

Listing 3.11: Deposit Function

EmergencyWithdraw The next step is the withdrawal of the funds. This function, ?? has the purpose of getting the deposited token back and setting `user.amount` and `user.rewardDebt` equal to 0. The fundamental variable `user.amountWithBonus` is still larger than 0. This is fundamental for the realization of the exploit.

```
// Withdraw without caring about rewards. BonusAmount is not updated
function emergencyWithdraw(uint256 _pid) external {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
}
```

Listing 3.12: Deposit Function

Withdraw In the last step, the attacker call this function with `_amount` equal to 0. The pending variable is estimeted, but the `user.rewardDebt` is equal to 0. Since both `pool.accRadsPerShare` and `user.amountWithBonus` are positve number, the result is larger than 0 as well. Since the amount is 0, the code cannot adjust the `user.amountWithBonus` variable to indicate the user claims the reward.

```
    function withdraw(uint256 _pid, uint256 _amount) external validatePool(_pid) {
      ...
        uint256 pending = user.amountWithBonus.mul(pool.accRadsPerShare).div(1e12).sub(user.
    rewardDebt);
        if(pending > 0) {
            if(pool.isSRadsRewards){
                safeSRadsTransfer(msg.sender, pending);
            }
            else{
                safeRadsTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            uint256 _bonusAmount = _amount.mul(userBonus(_pid, msg.sender).add(10000)).div
    (10000);
            user.amountWithBonus = user.amountWithBonus.sub(_bonusAmount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
            pool.lpSupply = pool.lpSupply.sub(_bonusAmount);
        }
        user.rewardDebt = user.amountWithBonus.mul(pool.accRadsPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.13: Deposit Function

The `user.amountWithBonus` increases every time the attacker starts from the step 1. This enables the attacker to drains more and more tokens in the process.

3.5.2 Properties

The estimation of users'rewards the vulnerability of the smart contract.

The malicious sequence of functions involves the call of `deposit`, `emergencyWithdraw` and `withdraw`. Therefore, the attacker could get back the same amount of deposited tokens, but with a higher amount bonus for the reward. The parameter `amountWithBonus` of the user struct, which keeps track of the amount and the bonus, just increases even if the user receives the reward and it is withdrawing.

The property is a post condition of the function `withdraw`. In the case the user is getting reawards, so the estimation of the paramter `pending` is greater than 0, the value of the parameter that keeps track of the bonus has to decrease.

$$\begin{aligned} if(pending > 0) \\ user.amountWithBonus < old\_value\_of(user.amountWithBonus) \end{aligned} \tag{3.6}$$

3.6 XSURGE on BSC Chain

The **XSurgeWeb**'s whitepaper provides a presentation of the ecosystem. It is described as a great DeFi investing idea based on proprietary pricing algorithms embedded in the Surge Token Variants' contracts. Surge Token Variants each have their own Market Maker, allowing them to trade continuously and outlast both centralised and decentralised exchanges. The strategy is to reward long-term holding by increasing a holder's claim of the backing asset. Each Surge Token utilizes a built-in contract exchange system that renounces the need for a traditional liquidity pool. Both assets are stored within the contract itself, rather than a liquidity pool pair of the backing asset to the token using a traditional market maker method for exchange and price calculation.

3.6.1 The explpoit

One of the Surge Token is SurgeBNB, the one which is my focus of analysis. **XSurgeBNB** explains in deep how the attack to this contract occured. The team claimed that the attacker had stolen $5 million in SurgeBNB through a backdoor vulnerability. XSURGE stated that a potential security vulnerability in the SurgeBNB contract was discovered on August 16th.

The attack is mabe by 4 main steps:

1. the attacker borrow 10,000BNB through flash loans.
2. Use all the BNB to buy SURGE. According to the current price, the attacker can buy 1,896,594,328,449,690 SURGE
3. He calls the "sell" function, for selling the obtained SURGE.
4. The sale function alters the data after the transfer, and the transfer code has a reentrance vulnerability. When the attack contract acquires BNB, the period before the SURGE contract's state changes (**??**), the attack contract can use the reentrance vulnerability to purchase SURGE again.

```
 function sell(uint256 tokenAmount) public nonReentrant returns (bool) {
     ...
     //The reentrancy
     (bool successful,) = payable(seller).call{value: amountBNB, gas: 40000}("");
     if (successful) {
         // subtract full amount from sender
         _balances[seller] = _balances[seller].sub(tokenAmount, 'sender does not have
this amount to sell');
         // if successful, remove tokens from supply
         _totalSupply = _totalSupply.sub(tokenAmount);
     } else {
         revert();
     }

     return true;
```

```
    }
```

Listing 3.14: Sell function of Surge (SURGE) token.

The bnb amount of the contract stays intact, and the total amount of SURGE tokens `totalSupply` has not been updated, because the attack contract spends all of the BNB balance to acquire SURGE each time (still remains the quantity before the sell). As a result, the price of token falls, allowing the attacker to purchase additional SURGE.

Repeating three times of Round 2 and Round 3 , the attacker accumulates a large amount of SURGE through reentry, and then sells all the SURGE to make a profit.

At the end of this transaction, the attack contract sold 1,864,120,345,279,610,000 SURGE, obtained 10327 BNB, and finally the profitable 297 BNB was sent to the attacker's address.

The following are the modifications suggested by the Beosin technical team for this attack:

- any transfer operation should be place after the state changes to avoid reentry assaults.
- Instead of using `call` use `transfer` or `send`.

3.6.2 Properties

This exploit represents a typical case of reentrancy.

The attacker's strategy involves the function sell, which contains the bug, and then the function purchase. After calling the first one and triggering the reentrancy, the malicious fallback implemented by the attacker uses the amount of money for buying more XSURGE tokens. At the end of the selling process, the total supply should decrease the amount sold by the user. But since the attacker called the purchase, the variable is not updated as it was supposed to be. Buying the same amount of sold tokens, the value would not change.

We define the property as a postcondition,refered to the function `sell`, which states the variable `_totalSupply` is decreased of the amount sold by the user, then `tokenAmount`.

$$\_totalSupply + amount == old\_value\_of(\_totalSupply) \tag{3.7}$$

The property can be even expressed like a an invariant, stating that the sum of the single balances cannot exceedes the variable _totalSupply.

$$sum\_of\_uint(\_balances) <= \_totalSupply \tag{3.8}$$

3.7 Reentering the Reentrancy Bug: Disclosing BurgerSwap's Vulnerability

BurgerSwap is an automated Marker Maker service on Binance Smart Chain (BSC). At time of the disclosure of the vulnerability, there was areound $13K worth of Ether at immediate risk. The vulnerability was was presentend by **BurgerSwap**.

It is a Binance Smart Chain fork of Uniswap, Automated Marker Maker (AMM) service operating on Ethereum. Tranding and listing Specialized BEP-20 tokens among standard swapping options are available on this platform. To mint such tokens, users can use BurgerSwap's "bridge" contract on Ethereum.

Brige is a combination of 2 smart contracts deeployed on different chains. It allows cross-chain transfers of value. Ether deposited into the contract on the main net will provide a balance denominated in ERC-20 tokens on the sidechain. While ERC-20 tokens deposited back into the contract on the sidechain can free up Ether on main net. One example could be locking Ether, which is converted via the contract to WETH (Wrapped Ether, an ERC-20 token pegged to Ether), and then the same wallet locking ETH can be credited with bWETH on BSC.

3.7.1 The exploit

The issue deals with the fucntion withdrawFromBSC, **??**. First of all, it checks some conditions and then it proceedes to transfer the amount to the mseeage sender. The order of the actions is:

1. It verifies executeMap[_paybackId] is false;
2. It checks _signature is a valid signature on _paybackId, _token, msg.sender, and _amount.
3. It calls TransferHelper.safeTransferETH(msg.sender, _amount).
4. It sets executeMap[_paybackId] to true.

The issue is the interaction with the sender's address (step 3) happens before the internal effect (step 4): reentrancy is feasible.

```

    function withdrawFromBSC(bytes calldata _signature, bytes32 _paybackId, address _token,
    uint _amount) external payable {
        require(executedMap[_paybackId] == false, "ALREADY_EXECUTED");

        require(_amount > 0, "NOTHING_TO_WITHDRAW");
        require(msg.value == developFee, "INSUFFICIENT_VALUE");

        bytes32 message = keccak256(abi.encodePacked(_paybackId, _token, msg.sender, _amount
    ));
        require(_verify(message, _signature), "INVALID_SIGNATURE");

        if(_token == WETH) {
            IWETH(WETH).withdraw(_amount);
            TransferHelper.safeTransferETH(msg.sender, _amount);
        } else {
            TransferHelper.safeTransfer(_token, msg.sender, _amount);
        }
        totalFee = totalFee.add(developFee);

        executedMap[_paybackId] = true;

        emit Withdraw(_paybackId, msg.sender, _token, _amount);
    }
```

Listing 3.15: BugerSwap Bridge vulnerable function

Folowing the execution of the code, the bug is found in the safeTransferETH function, contained in TransferHelper library. The expression `to.callvalue:value(new bytes(0))` is actually a call to the sender of the message, which can be an arbitrary smart contract. The malicious contract can implemnt a fallback function. By the time it receives the ether, the fallback function is triggered and `withdrawFromBSC` is run again, but without updating `executeMap[_paybackId]`. Since it is not set to true, the code repeat the same sequence of operation. Repeating this process within the same transaction, the attacker will drain the vulnerable contract's WETH holdings and credit.

3.7.2 Properties

The keyword of this exploit is reentrancy.

The smart contracts adopt an access control strategy at the beginning of the function, checking the signature of the user. The attacker forked MetaMask, a crypto-wallet, for allowing a smart contract to access the vulnerable function. However, it is out of our interest and we focus on the vulnerability in the code.

The vulnerable function is `safeTransferETH`, resonsable of sending ETH to the user. This is called by the one which manages the withdrawal of funds from the contract by the user. The malicous contract, thanks a fallback, can call multiple times the same function and withdrawing more money than it could.

With the postcondition, we check the balance of WETH (wrapped ETH) before and after the function, stating the difference should be the parameter amount of the function.

$$old\_value\_of(address(this).balance) == address(this).balance - \_amount \tag{3.9}$$

3.8 Infinite minting of NFTs: DirtyDogs NFTs

he project DirtyDogs implements a platform for buying and minting its collection of NFTs. It implements the specific standard ERC721 for managing NFTs.

The smart contract allows the users to buy the NFTs, before the official selling phase, buying a ticket, which would be used for minting those. The protocol should keep track of the ticket for minting the correct number of NFTs.

3.8.1 The exploit

DirtyDogs NFT contract has a typical example of reentrancy. The attacker exploited the function `claimDogs`, shown in ??. Firstly, the malicious wallet bought a ticket for having the right of receiving an NFT, calling the function claimDogs(). It loops on the number of tickets the sender has, and it calls the function _safeMint for creating the NFTs and sending them to the caller.

The bug involves the update of the variable `totalClaimed[_msgSender()]` at the end of the loop. It is the one which keeps track of the number of tickets owned by the caller.

The malicious smart contract implemented a callback function: the main trigger for reentrancy attacks. Within the same transaction, it gets the opportunity to execute the same code multiple times. When the smart contract receives an NFT, the fallback function is triggered and the `claimDogs` function is called again. As result, the attacker could call again the function for mintining, but without updating the variable which counts the number of tickets per address. The exploit produced 45 NFTs because the fallback has the risk of reverting, there is a limit of times to be called.

```
contract ERC721 is Context, ERC165, IERC721, IERC721Metadata, IERC721Enumerable {
        ...
    function _mint(address to, uint256 tokenId) internal virtual {
        require(to != address(0), "ERC721: mint to the zero address");
        require(!_exists(tokenId), "ERC721: token already minted");
        _beforeTokenTransfer(address(0), to, tokenId);
        _holderTokens[to].add(tokenId);
        _tokenOwners.set(tokenId, to);
        emit Transfer(address(0), to, tokenId);
    }
    ...
}

    ...
contract DirtyDogs is ERC721, Ownable {
    ...
    function claimDogs() external {
        uint256 numbersOfTickets = getUserClaimableTicketCount(_msgSender());

        for(uint256 i = 0; i < numbersOfTickets; i++) {
            uint256 mintIndex = totalSupply();
            _safeMint(_msgSender(), mintIndex);
        }

        totalClaimed[_msgSender()] = numbersOfTickets.add(totalClaimed[_msgSender()]);
    }

    function getUserClaimableTicketCount(address user) public view returns (uint256) {
        return presaleNumOfUser[user].add(publicNumOfUser[user]).sub(totalClaimed[user]);
    }
    ...
    }
```

Listing 3.16: DirtyDogs NFT contract

3.8.2 Properties

The attacker could exploit the smart contract due to a vulnerability in the function claim-Dogs, which includes even a bad implementation of the ERC721 standard. It can be classified as a case of reentrancy.

The NFTs are not directly sold, but a ticket instead is provided to the users, who would convert it. The function claimDogs is in charge to verify the tickets and generate the NFTs. Because of the reentrancy, the attacker produced 45 NFTs with a single ticket.

The property can be expressed as a postcondition, which states that the amount of NFTs produced should be equal to the number of tickets of the user.

We implemented even a support variable called effClaimed, which is increased for every minting, and the totalClaimed instead is updated just at the end of the function. The following post-condition is referred to as the vulnerable function claimDogs(...).

$$sum\_of\_uint(totalClaimed) <= effclaimed \tag{3.10}$$

4 Analysis Tools

This chapter is addressed to present the analysis tools involved in the experiments. Each one has a dedicated section, which gives a general overview of its architecture, the implemented methodologies and the limitations. We collected the information from their presenting paper or the provided GitHub documentation.

Our work deals with the practical aspect of the tools, their installation and their behaviour in a real-world case, consequently, we decided to stress this practical aspect for defining their taxonomy. We draw it, classifying the tools based on their running mode. **??** depicts our classification, involving three different subgroups:

- tools without specifictation;
- tools with specifictation;
- tools with multiple running mode.

The first one covers the ones which can be run with the source code (or bytecodes) of the smart contract without any additional data. Their strategies can detect a predefined group of vulnerabilities. On the other hand, the second group involves the ones which require the user provides specifications for running. The specification can involve a rule to be broken or a property which can be proved or unproved. In our taxonomy, the meaning of "multiple mode running" implies the tools which can be run with and without provided specifications.

The choice of these eight analysis tools is based on our literature research, which encompasses surveys and comparisons of those. Another discriminant aspect of the choice is the maintenance of the tools. the ones which have not been updated for more than two years were discarded Open-source tools are considered because their architecture descriptions and documentation are available.

Table 4.1: Tools Classification

Tools with Specifictation	Tools without Specifictation	Tools with Multiple Running Mode
SolcVerify	Mythril	Manticore
Celestial	Slither	SmarTest
Echidna		
Certora		

This thesis involves justt a tool which is not open-source, Certora. The aim is to have an experience of a mantained tool by a private company and not by the community, but still adopting the free version and the online documentation.

The objectives of the analyses are smart contracts involved in real-world exploits. The attackers mostly forced a bug involving the logic of the software itself. For this reason, we mostly picked tools which could even have custom analysis.

4.1 Tools with multiple mode running

This section has the aim to describe the tools classified as "with multiple modes running". Even other tools have more modes running, but this categorization involves the tools which could take part in the other two subgroups: with and without specification.

4.1.1 SmarTest

SmarTest is a safety analayzer for Ethereum smart contracts develeoped by **SmarTest**. It adopts a symbolic execution technique for effectively detecting vulnerable transaction sequences. The main challenge of the project involves the tool to find transaction sequences, revealing the vulnerabilities of the analysed smart contract. Therefore, bugs are discoved as the cause of the interaction of multiple transactions. The purpose of SmarTest is to automatically deliver vulnerable transaction sequences, which demostrate the weaknesses of the smart contract. The main idea is to build a statistical model using known vulnerable transaction sequences and use it to direct symbolic execution toward more successfully detecting unknown vulnerabilities. Symbolic execution is guided by statistical language models, so it can prioritize transacion sequences which are likely to reveal vulenrablities. This statregy involves firstly to run unguided symbolic execution on existing vulnerable contracts, then to learn a probablity distribution over vulnerable transaction sequences.

The tool is implemented as an extension of VeriSmart (**VeriSmart**). SmarTest is build on top of that, adding its own functionalities:

- symbolic execution with a language model.
- Symbolic executor for transaction sequences.
- Constraint solving optimization.

The installation of VeriSmart is necessary for running the tool. After that, VeriSmart can be used in SmarTest mode.

The report **??** shows an example of output of SmarTest, which provides the sequence of funtions for exploiting the found bug. It states in which line the Vulnerability is, then it provides a list of transacion which could force the it.

```
[5] [IO] line 39, (balance[_to] + _value) : disproven, 14.528264s
1: Example
   {}
   {msg.sender: #x0000000000000000000000000000000000010000,
    msg.value: 0}
    2: approve
       {_spender: #x0002000000000000000000000000000000000000,
        _value:
    44365792925664701906080996193724747326645573793336555789802397725137091694592}
       {msg.sender: #x0000000000000000010000000000000000000000,
        msg.value: 0}
    3: mintToken
       {_target: #x0000000000000000010000000000000000000000,
        _amount:
    87371285831589357636669861644764241805818792173739087408632338890371299803136}
       {msg.sender: #x0000000000000000000000000000000000010000,
        msg.value: 0}
```

Listing 4.1: SmarTest Example Report.

The tool implements modules for the detection of the following vulnerabilities:

- integer over/underflow,
- assertion violation,
- division-by-zero, ERC20 standard violation,
- Ether-leaking vulnerability (e.g. unauthorized access to transfer),
- suicidal vulnerability (e.g., unauthorized access to selfdestruct).

In the paper, the authors focus on just those, without considering vulnerabilities that require analysis of the interaction of multiple contracts to demonstrate the flaws (e.g., reentrancy).

4.1.2 Manticore

ManticorePaper describe in their paper an open-source dynamic symbolic execution framework called Manticore for analyzing binaries and Ethereum smart contracts.

The definition of dynamic symbolic execution is:

Definition 2. *Dynamic symbolic execution mixes concrete and symbolic execution benefiting from the efficiency and decidability of concrete execution and the need to use concrete values when interacting with the program environment and from the stronger guarantees of symbolic execution.*

The adaptable architecture of Manticore enables it to run custom analsysis, and its API increase the customization of those.

The keyword of this tool is dynamic symbolic execution, which is the implemented analysis technique. It identifies a collection of path predicates, and constraints on the program's input, for paths that the analysis has investigated. These are employed to produce the inputs necessary for the corresponding paths to be followed.

?? provides an overview of Manticore's architecture. The primary components are the Core Engine and Native and Ethereum Execution Modules. Secondary components include the Satisfiability Modulo Theories (SMT-LIB) module, Event System, and API.

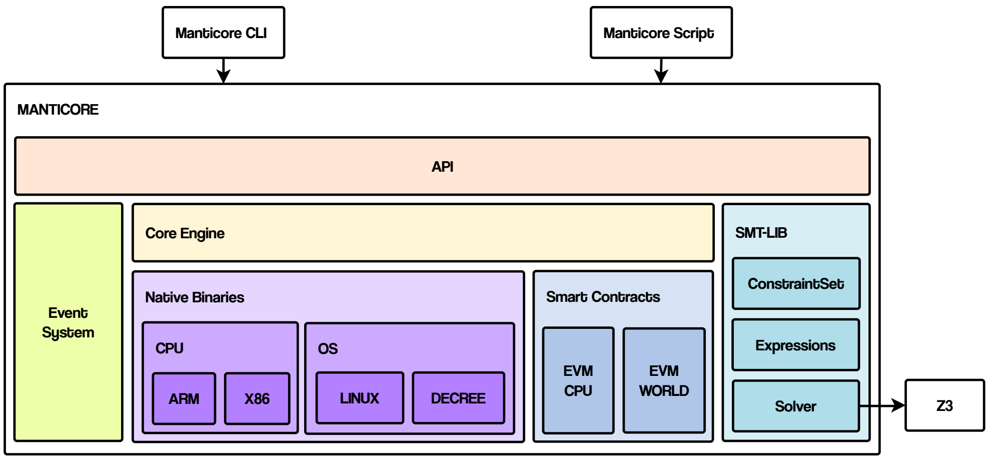

Figure 4.1: High-level architecture diagram (Figure from [**ManticorePaper**]).

Manticore's adaptability comes from the Core Engine. It implements a generic platform-independent symbolic execution engine with few underlying execution model presumptions. Despite the differences between EVM and no EVM execution, Ethereum support did not require substantial architectural changes to Manticore, since the Core Engine is completely decoupled from all execution platform details.

Dealing with Symbolic execution of smart contracts, those get input as network transactions consisting of a value and a data buffer, containing information about which function should be executed in a contract, and its arguments. This technique involves symbolic transactions, where both value and data are symbolic. Symbolic transactions are applied to all Ready states, which cause the symbolic execution of one transaction. This allows the tool to explore the state space of a contract. An infinite number of interconnected contracts can be executed in Manticore's emulated environment. It can monitor the condition of not just a single contract but the entire Ethereum "world," which consists of many interconnected contracts.

The specifications are written in terms of functions. These do not have any parameters and return a boolean statement. The tool in the report states which function could not pass the test, which means that the function returned false. It indicates even the list of transactions for making the function return false, so "breaking" the specified property.

On the other hand, it can run without specfications. It implements plugins, which can be enable or disable, for detecting vulnerabilities. It can detect the following ones:

- integer overflow/underflow,
- reentrancy,
- unused retturend value (which is not a proper vulnerability, but the tool warns the user of this case),
- suicidal contracts,
- detect delegatecall (the tools warns the user of this case),

- race condition.

4.2 Tools with Specifications

This section deals with the description of tools with specifications, so the ones which need the source file even additional information. Some of the tools requires just some modifications of the source code, on the other hand in some cases the user has to provide an additional file providing the specifications.

4.2.1 Certora Prover

Certora Prover is a tool for formal verification of Solidity smart contracts. Any computer programme that may be compiled using EVM can undergo Certora Prover verifications.

Since the tool is not open source, we draw the information regarding the tool from the **CertoraDocumentation**

It is provided as Software as a Service, a cloud technique, so it is not possible to install the complete tool. A user can interact with it on its website, providing the tool as SaaS (service as a Service), otherwise, a command-line interface can be downloaded, which interacts with the server by remote.

The user for the verification has to provide the Solidity file and the specification one, which contains the logic formulas for verification conditions. These are proven by an SMT (satisfiability modulo theories) solver. The solver also provides a concrete test case demonstrating the violation of the specifications that are unproved.

A user can state the specifications as invariants otherwise as functions, called rules. The invariants contain just a boolean formula. On the other hand, the rules contain the properties to be proved and these are written like functions. Those are written using the Certora language, which has a similar grammar to Solidity, for example the types of the variables are the same, but with more functionalities for writing down the specifications. The specification file allows the call the function of the target smart contracts- Multiple `require` definitions can be stated at the beginning, for expressing a condition to be realised before the running of it. A requirement is that each rule has to conclude with `assert`, containing a boolean condition. The user can specify an enviroment variable, `env` defining the possible preconditions.

?? covers an example of the specification of the function `transferFrom` of a smart contract which implements a token. The function cares about the transfer of tokens from one accout to the other one. This rule checks that the balances of the users are updated correctly. The enviroment of the enviroment specifies that the `msg.value`, so the amount of native token sent in the transaction, is 0.

```

rule transferFromCorrect(address from, address to, uint256 amount) {
    env e;
    require e.msg.value == 0;
    uint256 fromBalanceBefore = balanceOf(from);
    uint256 toBalanceBefore = balanceOf(to);
    uint256 allowanceBefore = allowance(from, e.msg.sender);
```

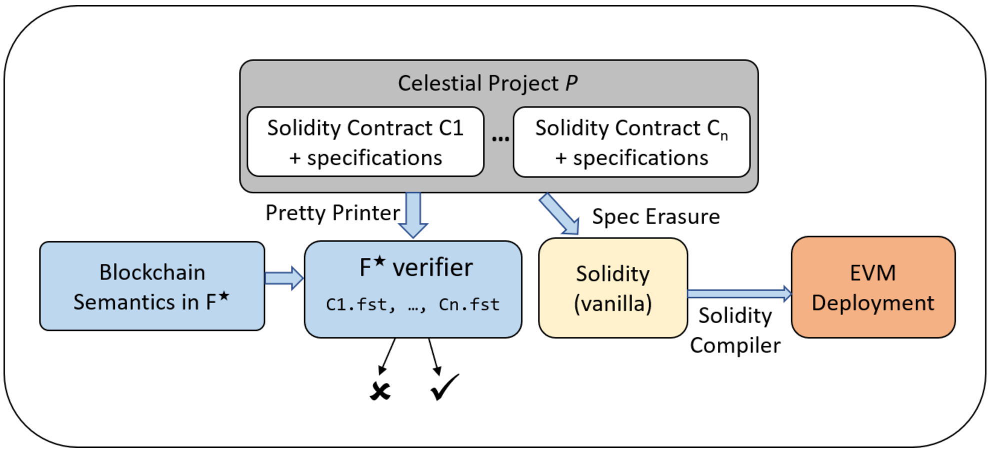

Figure 4.2: Celestial Architecture

```
    require fromBalanceBefore + toBalanceBefore <= max_uint256;

    transferFrom(e, from, to, amount);

    assert from != to =>
        balanceOf(from) == fromBalanceBefore - amount &&
        balanceOf(to) == toBalanceBefore + amount &&
        allowance(from, e.msg.sender) == allowanceBefore - amount;
}
```

Listing 4.2: Certora example specifications

4.2.2 Celestial

This subsection is addressed to introduce Celestial, an analysis tool for Solidity Ethereum-based smart contracts developed by the research team of Microsoft India. ?? shows the its architecture.

The developers provide functional requirements for formally verifying their specifications. The input file is labelled It gives programmers the ability to create functional requirements for their contracts. The input file is labelled as ".cel", it is the solidity file, with the added specification expressed in notes. When the grammar is checked, the contract and the specifications are translated in F* for having the verified verdict.

?? shows an example of input file. The invariants are expressed as boolean condition. At the beginning of a function, the specifications can be expressed, regarding precondition and postcondition. The keyword `modifies`, placed as well at the beginning of the functions, specifies the variable that can be modified in the function, or `tx_reverts`, which states the possible condition that a function can revert. The Solidity implementation of the function is kept.

```
    contract SimpleMarketplace {
        // contract fields
        invariant balanceAndSellerCredits {
            balance >= totalCredits &&
            totalCredits == sum_mapping ( sellerCredits )
        }
        //function
        function buy ( address itemId ) public
            modifies [ sellerCredits , totalCredits , itemsToSell ,
            log ]
            tx_reverts !( itemId in itemsToSell ) || value != itemsToSell [ itemId ].price
            || value + totalCredits > uint_max
            post (!( itemId in itemsToSell ) && sellerCredits [ seller ] == old (
    sellerCredits ) [seller = > sellerCredits [ seller ] + value ]
            && log == ( eItemSold , sender , itemId ) :: old ( log ) )
         { // implementation of the buy function }
    }
```

Listing 4.3: Celestial example specifications

F* is a fully dependent type system proof helper and programs verification. The authors state that F* offers SMT-based automation, which is sufficient for the completely automated verification of real-world smart contracts. Moreover, it enables the developers to work in a customised state and exception effect mimicking the blockchain semantics since it supports user-defined effects. Finally, even though its first-order subset with quantifiers and arithmetic is used by the tool, it permits expressive higher-order specifications.

The celestial process involves 2 steps: the translation of the specification and the verification of F* start. The first one involves a python script, on the other hand, the second one entails the installation of F* engine. The output covers the response of the verification and a generated solidity file, which represents the smart contract without the specifications notes.

Limitations The authors explained their tool implementation focused on the Solidity constructs used in their case studies, therefore it does not cover some Solidity cases.

Delegatecall, embedded assembly It does not take into account syntactic elements like inheritance, abstract contracts, tuple types, delegatecall and embedded assembly

Most of these only offer syntactic sugar, which CELESTIAL's future iterations should find simple to support. Arrays and structs cannot presently be passed as parameters to functions in our implementation.

Loops are allowed in the smart contracts, however, the tool does not support loop invariants. When external contracts are called, reentrant behaviour can result, in which the external contract contacts the caller back. Reasoning about reentrant actions is frequently counterintuitive. Celestial forbids these actions, this property is called "external callback freedom" (ECF). It states that every callback execution in a contract is equivalent to some activity without reentrancy. So Celestial assumes that there is no callback during the external call.

4.2.3 Echidna

Echidna is an open-source smart contract fuzzer, developed by **Echidna**, which makes it easy to automatically generate tests to detect violations in assertions and custom properties.

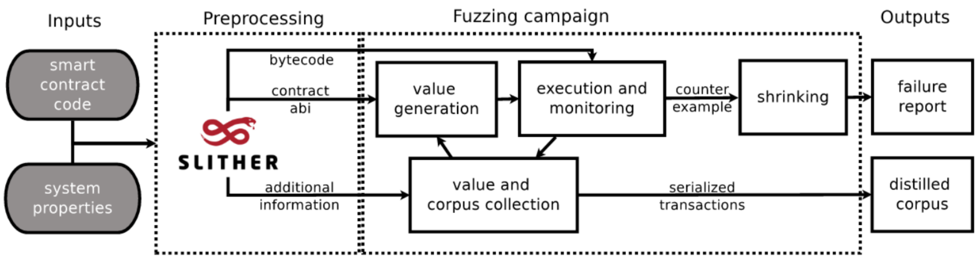

Figure 4.3: Echidna architecture (image from [**Echidna**]).

Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities during fuzzing campaigns, Echidna supports three types of testing:

- user-defined properties (for property-based testing);
- assertion checking;
- gas use estimation.

Figure **??** depicts the Echidna architecture as a two-step process: pre-processing and fuzzing. The tool starts with a collection of contracts that have been supplied, as well as attributes that have been integrated into one of the contracts. Echidna uses Slither, smart contract static analysis framework presented in **??**, to build and analyse the contracts to find relevant constants and functions that directly handle Ether (ETH). The fuzzing effort begins in the second stage. Using the application binary interface (ABI) given by the contract, significant constants stated in the contract, and any previously gathered sets of transactions from the corpus, this iterative procedure creates random transactions. When a property violation is detected, a counterexample is created to indicate the smallest and most basic sequence of operations that caused the failure.

The code **??** provides an example of invariant in Echdina context. The Solidity contract contains a vulnerability a the backdoor function. The output of the terminal presents the subsequence of functions to call for breaking the rule.

```
contract Token{
    mapping(address => uint) public balances;
    function airdrop() public{
        balances[msg.sender] = 1000;
    }
    function consume() public{
        require(balances[msg.sender]>0);
        balances[msg.sender] -= 1;
    }
    function backdoor() public{
        balances[msg.sender] += 1;
    }
    function echidna_balance_under_1000() public view returns(bool){
        return balances[msg.sender] <= 1000;
    }
}
```

Listing 4.4: Echidna usage example.

The tool can be even used to test assertions. The aim is equivalent of the invariant testing methodology, but in this case properties are expressed using the Solidity annotation of assertion.

4.2.4 Solc-Verify

SolcVerify present Solc-Verify, a source-level verification tool for Ethereum smart contracts. It takes smart contracts written in Solidity and discharges verification conditions using modular program analysis. It is built on top of the Solidity compiler, so it reasons at the level of the contract source code. Because of that, Solc-verify can reason about high-level contract attributes while accurately modelling low-level language semantics.

Solc-Verify is implemented as an extension to the Solidity compiler. It accepts a collection of Solidity contracts, including specification annotations, and uses the Boogie verifier and SMT solvers to discharge verification conditions.

As **SolcVerify_2** explain, Solc-verify translates the annotated contracts to the Boogie Intermediate Verification Language (IVL). The tool relies on the Boogie verifier to perform modular verification by discharging verification conditions to SMT solvers. The verification conditions encode the function body while assuming the preconditions, and then check if postconditions hold. In this process, function calls are replaced by their specification and loops by their invariants (modularity). Finally, the results are back-annotated to the Solidity source.

?? present an example of annotation, which states that the contract will ensure that the sum of individual balances is equal to the total balance in the bank.

```
pragma solidity >=0.7.0;

/**
 * @notice invariant __verifier_sum_uint(balances) <= address(this).balance
 */
contract SimpleBank {
    mapping(address=>uint) balances;

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

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] > amount);
        bool ok;
        (ok, ) = msg.sender.call{value: amount}(""); // Reentrancy attack
        if (!ok) revert();
        balances[msg.sender] -= amount;
    }
}
```

Listing 4.5: An example Solidity smart contract implementing a simple bank with SolcVerify annotations.

SolcVerify_3 on GitHub repository, present the specification annotations. Those must be included in special documentation comments (/// or /** */) and must start with the special doctag @notice. They must be side-effect free Solidity expressions (with some verifier-specific extensions) and can refer to variables within the scope of the annotated element. Functions cannot be called in the annotations, except for getters. The currently available annotations are listed below.

- Function pre/postconditions can be attached to functions. Preconditions are assumed before executing the function and postconditions are checked (asserted) in the end. The expression can refer to variables in the scope of the function. The postcondition can also refer to the return value if it is named.
- Contract level invariants can be attached to contracts. They are included as both a pre and a postcondition for each public function. The expression can refer to state variables in the contract (and its balance).
- Loop invariants can be attached to for and while loops. The expression can refer to variables in the scope of the loop, including the loop counter.
- Modification specifiers can be attached to functions. The target can be a (1) state variable, including index and member accesses or (2) a balance of an address in scope. Notes, however, that balance changes due to gas cost or miner rewards are currently not modelled.
- Event data specification can be attached to events that should be emitted when certain data changes. Events can declare the state variable(s) they track for changes, or in other words, the variables for which the event should be emitted on a change.

4.3 Tools without specification

We introduce the second main category of our taxonomy: tools without Specifications.

These do not need any specifications and they based their analyses on preselected strategies. They can be considered "plug & play", indeed they work with code (or even bytecode) of the smart contracts, without the developer should code anything for letting them run. Most of the tools that we found during the literary research propose just a warning, depicting the vulnerable lines of code and classifying the vulnerability following their taxonomy or open-source ones.

4.3.1 Slither

Slither is described by **Slither** as an open-source static analysis framework. It uses its own intermediate representation, SlithIR, which was created to simplify static analysis of Solidity code. Concolic analysis, taint analysis, and control flow checking are involved for detecting a variety of security vulnerabilities. It is designed to provide granular information about smart contract code and the flexibility necessary to support many applications.

It is mainly used for:

- Automated vulnerability detection: a large variety of smart contract bugs can be detected without user inter- vention.

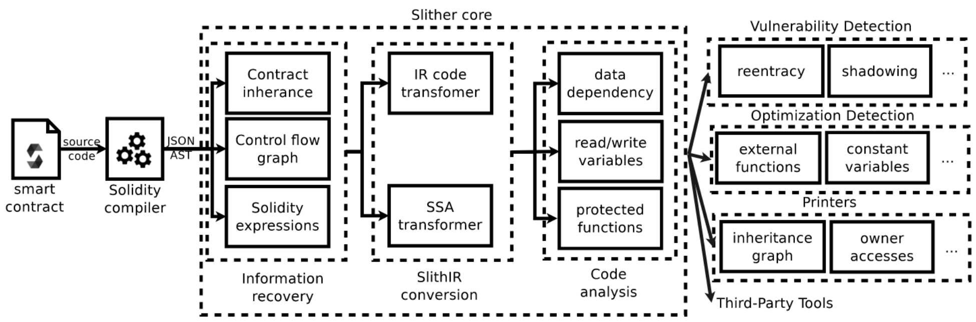

Figure 4.4: Slither architecture overview (image from [**Slither**]).

- Automated optimization detection: Slither detects code optimizations that the compiler misses.
- Code understanding: printers summarize and display contracts' information to aid in the study of the codebase.
- Assisted code review: through its API, a user can interact with Slither.

Slither implements more than twenty bug detectors, regarding reetrancy, Uninitialized variables, Shadowing and many other. The tool allows the developers to integrate more detectors, therefore it extends Slither's capabilities to detect more advanced bugs.

SlitherGitHub is written in python 3 and it is published on GitHub. During the installation, I did not find any particular issues.

4.3.2 Mythril

Mythril is a security analysis tool for Ethereum smart contracts. It was introduced by **Mythril**.

The tool relies on taint analysis and control flow checking of the EVM bytecode to prune the search space and look for values that allow exploiting vulnerabilities in the smart contract. It is targeted at finding common vulnerabilities and is not able to discover issues in the business logic of an application. **SWCRegistry**'s taxonomy of vulnerabilities is used by Mythril to classify them. **??** illustrates an example of the output of Mythril analysis. In the second line, there is a reference to the vulnerability classified by the SWC Registry with the ID of 110 (Assert Violation).

```
==== Exception State ====
SWC ID: 110
Severity: Medium
Contract: Token
Function name: transferArray(address[],uint256[])
PC address: 4385
Estimated Gas Usage: 944 - 6585
An assertion violation was triggered.
It is possible to trigger an assertion violation. Note that Solidity assert() statements
    should only be used to check invariants. Review the transaction trace generated for this
     issue and either make sure your program logic is correct, or use require() instead of
    assert() if your goal is to constrain user inputs or enforce preconditions. Remember to
    validate inputs from both callers (for instance, via passed arguments) and callees (for
    instance, via return values).
--------------------
In file: test.sol:309

function transferArray(address[] tos, uint256[] values) public returns (bool) {
        for (uint8 i = 0; i < tos.length; i++) {
            require(transfer(tos[i], values[i]));
        }

        return true;
    }

--------------------
```

Listing 4.6: Example of the output of Mythril Analysis.

5 Results presentation and Evaluation

This chapter presents the results of the analyses. It is addressed to show the obtained data from the tests.

The first part deals with the individual outcomes per tool, providing details about the running phase and installation one. Eventual problems, during those phases, are described per each tool.

Then, a general comperison gives an high level view of the obtained results. This section aims to define which tool had a better behaviour in term of operability, speed and correctness of the analysis.

5.1 Individual Outcomes per Tool

This section deals with the eventual problems during the installation and running phase , moreover, the tools outcomes having as objective a specific attack are collected. Per each tool, a table has three parameters; the first one is the number of written lines of code considering the specifications. The part of the code which were copied from the target smart contracts were discarded during the counting. The second field deals with the execution time of the tool and the last one specifies if the vulnerability was effectively detected.

The code of the exploits was modified to keep the vulnerable aspects, reproducing their logic, but the irrelevant parts were discarded.

We kept the same simplified version of the smart contracts for all the tools.

Manticore The provided guide on Github ([**ManticoreGitHub**]) gives a detailed guide for Manticore installation. Since it is written in python, we used a virtual environment and we counted around nine libraries for dependences. The installation involved basically one command, since it was managed by "pip", python packet manager. We adopted its default running mode for the attacks involving reentrancy and "Manticore-verifier" running mode for the others.

The table **??** shows the outcames of the analyses per attack; the symbol "−" means that in that case we did not use the "Manticore-verifier" running mode, but the default one, which it does not need a specification file.

SmarTest SmarTest is built on top of VeriSmart tool, so it can be seen like a plug in of these one. Indeed, we run SmarTest as an option of VeriSmart, as the Github guide explains ([**SmarTestGitHub**]). The tool is built with OCalm, a program language, so we used for

Table 5.1: Manticore results

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud
Aku	6	235	Not Found
Cover	5	245	Not Found
BZX	4	228	Found
Spartan	3	239	Found
Uranium	3	250	Not Found
XSURGE	–	208	Found
BurgerSwap	–	205	Found
DirtyDogs	–	203	Found

Table 5.2: SmarTest results

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud
Aku	3	320	Found
Cover	2	310	Not Found
BZX	4	320	Found
Spartan	2	320	Not Found
Uranium	4	320	Found
XSURGE	–	–	Not Found
BurgerSwap	–	–	Not Found
DirtyDogs	–	–	Not Found

the installation "opam", which is a source-based package manager for it. An important dependency is Z3, its satisfiability modulo theories (SMT) solver. Solc, the compiler for solidity, is required.

Since it has no the dector for reentrancy, as the table **??** shows, the attacks involving it were discarded. We adopted just the "assertion" running mode, for obtating valuable results. We fixed a running time threshold of 320 seconds.

Celestial Celestial is a tool which encompasses two main steps: the translation of the target smart contract in F* and then the running of the formal verification engine. This is the tool that required the most amount of time for installation and usage. It did not cover reentrancy attacks, so those exploits were discarded, moreover, it did not cover the keywords "storage" and "memory", so the "Cover Protocol" exploit was discarded as well. **??** shows the tool could detect the other four vulnerabilities. During the running, we used multiple versions of F* and in some cases (as with the "Uranium" exploit) the conversion was not correct so we adjusted the F* code to let it work.

Table 5.3: Celestial results

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud
Aku	22	4	Found
Cover	–	–	Not Found
BZX	15	3	Found
Spartan	29	5	Found
Uranium	18	5	Found
XSURGE	–	–	Not Found
BurgerSwap	–	–	Not Found
DirtyDogs	–	–	Not Found

Table 5.4: Echidna results

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud
Aku	7	22	Found
Cover	5	7	Found
BZX	3	33	Found
Spartan	3	17	Found
Uranium	3	24	Found
XSURGE	–	–	Not Found
BurgerSwap	–	–	Not Found
DirtyDogs	–	–	Not Found

Echidna Echidna is a fuzzer for smart contracts. The tool did have any problem during the installation and worked fluently. It allows to write the properties in form of function or as statement of "assertion". Both solution were applied depeing on the case. For "BZX" and "Spartan" exploits assertion mode was, the other one for the other cases. **??** states that the tool could detect all the vulnerabilities, but the ones involving external calls, since it discards those.

Certora Since Certora is not an open source tool, the provided documentation is limited. Their GitHub ([**CertoraGitHub**]) has a repository dedicated to tutorials for understanding how to write down the properties. **??** states that the tool could detect all the vulnerabilities, but the ones involving external calls, since it discards those.

SolcVerify SolcVerify deals with formal verification for smart contracts. Its annotation language does not require a great amount of lines of codes. For working properly, as Celestial, all the involved functions requires specifications. Installation problems occured

Table 5.5: Certora results; the time is provided by the sas application

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud
Aku	52	14	Found
Cover	31	21	Found
BZX	25	18	Found
Spartan	20	25	Found
Uranium	42	27	Found
XSURGE	–	–	Found
BurgerSwap	–	–	Found
DirtyDogs	–	–	Found

Table 5.6: SolcVerify results

Attacks	Lines of Code	Execution Time (seconds)	Foud, Not Foud (seconds)
Aku	9	4	Found
Cover	13	5	Found
BZX	17	9	Found
Spartan	25	17	Found
Uranium	23	9	Found
XSURGE	20	10	Found
BurgerSwap	11	10	Found
DirtyDogs	30	14	Found

during the right configuration .NET and the selection of the right version of the external dependences. Moreover, the tool had issues for finding the correct path of those in the system.

?? shows the tool could detect all the vulnerabilities.

Slither and Mythril Slither and Mythil did have any problem during the installation phase and run properly per each attack. They could detect all the reentrancy issues, but none of the others.

5.2 General Comparison

The first step was the collection of all the data from the analyses with the vulnerable smart contracts as objective. In this part, we provide a technical comperison of the tools.

5.2.1 Installation

?? collects the info about the installation detalais per each tool.

Table 5.7: Installation and running mode

Tools	Running modes	Extenral Dependences	OS
Manticore	2	2	Linux, OS X
SmartTest	2	3	Linux, OS X, Windows
Celestial	1	3	Linux, OS X
Echidna	2	2	Linux, OS X, Windows
Certora	1	–	–
SolcVerify	1	4	Linux, OS X
Mythril	1	2	Linux, OS X
Slither	1	2	Linux, OS X, Windows

The running mode involves the way a tool can be run, a tool is classified as "multiple running mode", if its grammar effectivly changes between different modes. For example echidna, it can be run in test mode and in assertion one, in the first case it requires functions with boolean formulas, in the second case the solidity keyword "assertion" in the code.

The external dependences encompass the exteranal elements, so we do not count the amount of libraries of the same language. All the tools require Solc, the solidity compiler. Most of the tools did not have any problem during the installation phase. Celestial and SolcVerify installations had issues. The first one needs F* for running and the selection of its right version is based on the tool and the system environment, consequently, it turned like a challenge. On the other hand, the second one involves the usage of Boogie, an intermediate verification language, which requires .NET, ([**NET**]), an open source, cross-platform for building many kinds of applications mantained by Microsoft. The configuration of the correct version of .NET and the other external dependences was based on the computer enviroment and the tool.

Certora operates in Service as a Service (SaaS) mode, so it can be classified as the most compatible tool, but it requires to be connect to internet. SmartTest can be run in two modes, but it was not necessary, since the running mode without specifications could not provide any consistent results.

5.2.2 Outcomes

In this subsection, the outocomes of the tools are compared. **??** has the name of tools as rows and the real-world exploits as columns. It shows which tool was able to scan the specified vulnerability involved in the attacks. The caption states the symbols used for the evaluation:

- ✓states that the vulnerability was scanned;
- ✗means the tool was no able to detect it;
- – stands for "discarded", tha attack was not considered for architectural reason of the tool.

Table 5.8: Analyses Outocomes per Attack: ✓: Found vulenrablity, ✗: Not found vulnerability, −: Discarded

Tools	Aku	Cover	BZX	Spartan	Uranium	XSURGE	BurgerSwap	DirtyDogs
Manticore	✗	✗	✓	✓	✗	✓	✓	✓
SmartTest	✓	✗	✓	✗	✓	−	−	−
Celestial	✓	−	✓	✓	✓	−	−	−
Echidna	✓	✓	✓	✓	✓	−	−	−
Certora	✓	✓	✓	✓	✓	−	−	−
SolcVerify	✓	✓	✓	✓	✓	✓	✓	✓
Slither	✗	✗	✗	✗	✗	✓	✓	✓
Mythril	✗	✗	✗	✗	✗	✓	✓	✓

We can state that the tools with specifications could not detect reentrancy in most of the cases. SolcVerify is the only one which could provide a result in all cases, moreover, it was the only tool which could detect correctly all the vulnerabilities we provided.

Celestial is the tool with more discarded cases. The tools without specification could not detect any of the attacks, but the ones involving external calls.

The tools without specifications gave warning in most cases, which can be used as a hint for scanning the vulnerabilities. Echidna and Certora had a similar behaviour, they could not detect the attacks regarding reentrancy.

?? provides high level detailes about the analyses per tool. "Constructive output" is referred to the type of output of the tool. We have three types:

1. list of functions: the sequence of operations to execute for forcing the vulnerability is expressed;

2. list of unproved tests: the output provides only the name of the failed test;

3. warnings: the tool displays just warnings.

SolcVerify and Celestial, within the tools allowing custom analyses, are the only ones which do not provide any additional information. Manticore and Mythril display, when it is possible, the list of functions, but in the case of reentrancy just a warnings of possible risk.

Reagading the speed of the tools, Celestial tets times are assumed with zero delay between the generation of the F* code and the running of the F* module. The time spent for running by command line the two different operations could be avoided by automating the process. The slowest tools are the ones involving symbolic execution. Myththil needs a similar amount of time as Manticore and SmartTest, even if it does not allow custom analyses. Slither is fastest one, but specifications are not provided per test and no additional information are displayed about the scanned vulnerability.

Table 5.9: Analyses Outcomes: LoF: List of functions, LoU: List of unproved tests, W: Warnings

Tools	Constructive output	Avg lines of code for test	Avg time (in seconds)
Manticore	LoF, W	4	239,5
SmartTest	LoF, W	2,5	318
Celestial	LoU	21	4
Echidna	LoF	4	20,5
Certora	LoF	34	21
SolcVerify	LoU	18,5	10
Mythril	LoF, W	–	221
Slither	W	–	3,5

Within the tools with specifications, another parameter for the comparison is the number of lines of codes required for defining the specifications. The ones which required the least number of lines are the ones which allowed those definitions by using "assertion", so SmartTest and Echidna. However this approach does not have consistent results, Echidna worked better using the other mode.

Echidna and Manticore have similar results since they adopt the same grammar for expressing the specifications. Those are written in the form of a function which returns a boolean value.

Certora had the most amount of lines. A reason for that is its elasticity for defining specifications in terms of functions. SolcVerify, with the formal verification tools, had the best behaviour in this case, because of its annotation language. Considering the spefication files entirely, Celestial had the longer ones, because the specifications are written on the code.

6 Discussion

The aim of this chapter is to interpret and explain the obtained results. It provides insights on the found threats, having an high level overview. Same approach about the tools, which are grouped based on their startegies, for focusing their approach and understaning their behavior based on the exploits.

6.1 Threats in real-world exploits

The approach for the choice of the attacks covers their effectiveness in terms of created damage. It is estimated based on the stolen amount of money and the possibility to recover the lack of security. The targets of those are liquidity pools, autonomous market makers or NFTs markets. The decentralised finance investments are driven by the community, therefore, different trends can explode. Consequently, the attackers adapt their target based on that.

A class of attacks involves external calls, particularly reentrancy issues. One of the most cited attacks of this type is the DAO attack that happened in 2016. Since that, developers should verify problems regarding external calls, but still, nowadays those could happen.

The other exploit deal with problems regarding the sending of possible compensation to users. "Uranium", "Spartan" and "Cover" exploits are based on the manipulation of the estimation of rewards. The attackers could increase the rewards for withdrawing a greater amount of liquidity from the pool. Those attacks had different methodologies but the same aim.

The NFTs markets, involved in "Aku" and "DirtyDogs" exploits, were attacked in different ways. In the first case, the attack can be classified as denial of service, since the contract was stuck and could not refund the other participants of the bid. On the other hand, in the second case, the adopted strategy for distributing NFTs with tickets was forced for minting multiple of those. A bad implementation of libraries brought to have that vulnerability.

It comes clear as the threats had different origins based on the contracts. Reentrancy is still feasible, even if it is a well-known risk.

The first step in the prevention of attacks is the understanding of the logic of the target contracts. Possible attacks can involve the distribution of possible rewards or the process of minting tokens or NFTs with the usage of additional data structures.

6.2 Tools with Specifications

The tools with specifications allow the users to customize the analysis. In this section, those are grouped based on their startegies.

The considered tools cover the following security approach:

Table 6.1: Outcomes based on strategies

Strategy	Execution Time (seconds)	Found percentage	No reentrancy
Fuzzing	20,60	63%	100%
Formal Verification	11,67	70%	93%
Symbolic Execution	272,31	44%	31%

1. fuzzing;
2. symbolic execution;
3. formal verification.

?? depicts the outcomes of the analysis, grouping the tools based on their strategies. This approach acilitate the comprehension of the behavior of the tecniques in this field.

Formal verification is a very powerful security approach, intending to prove or unproved the given specification. This perfectly fits with our research goals. We involved three different tools, in implementing this approach, for our purpose.

Certora is the only one which provides a complete list of functions for breaking the rules, rather than just a warning. On the other hand, SolcVerify could detect the vulnerabilities involving external call functions, indeed reentrancy. A powerful aspect of this tool is its possibility to express loop invariants, the other ones do not allow it. Considering the grammar for expressing the specifications, SolcVerify is the one which needs the least amount of lines of code, indeed it involves a notification language.

Certora is the only tool which is not open-source, for our purpose we adopted its free version. Its specification language is described by its developers' group as "rule-based". It differs from the other two tools under this aspect, because this way gives more elasticity to the user and defines more specific cases. The rule is composed of some function calls and it concludes with an assertion or more. The user is allowed to test a specific case, using "require" and the possibility to set up a proper environment.

Celestial and SolcVerify needed specifications for all the functions for working properly.

Echidna is the fuzzer and it had similar results to the formal verification tools. It had a bit worse performance in terms of speed and amount of detection. If the exploits with external calls are discarded, this tool could detect all of those. It provides the list of functions for forcing the exploit. Since it is based on random inputs, we had cases that run at different times, and the inputs of the function change per time.

Symbolic execution is the approach which required the most amount of time. Manticore had a better approach than SmartTest because it allow the detection of malicious external calls. The tools had a fixed time out for not wasting time, in some cases it expired.

In terms of effectiveness and speed, formal verification had the best results. A strength of the fuzzer and the symbolic execution tools is their outcomes, which display even how the attack is computed. It could help the developer for fixing the bug.

Considering the exploits without malicious external calls, the percentage of detections changes, as shown by **??**.

Echdina could detect all the vulnerabilities in this case. Symbolic execution effectiveness decreased, on the other hand, formal verification increased. Those results depict that in the case of smart contracts involving external calls, symbolic execution could provide effective analysis. The other two approaches are more effective in the case of custom analysis based on specifications, so the detection of exploits involves breaking certain rules.

6.3 Customized and Non-specific Analyses

The attackers exploited a specific bug or lack of security in the logic of programs. For scanning those, the involved tools adopted different strategies, allowing custom analysis.

The specifications allowed the user to express the requirements of the program. The accuracy of the analysis depends on the definition of the specifications. Their correctness is fundamental for the effectiveness of the results. The definition of those represents a challenge, moreover, each tool has different rules and language for expressing those. The tools without specification implement automatic detectors. Their presentation papers, or documentation, specify which known vulnerabilities can detect. These detect a specified set of vulnerabilities. The users should consider this aspect during the analysis.

Tools of this group, as our results demonstrate, do not have the capability of scanning vulnerabilities involving the logic of the programs. However, those obtained consistent results regarding the reentrancy cases. An example is Slither, which for every comparison of block time stamp gives a warning. However, developers intentionally consider this case and develop it considering the risks.

On the other hand, the ones with specifications mostly discarded the malicious external calls. SolcVerify was the only tool which could provide the possibility of reentrancy detection. Echidna, as Certora, developers teams specified the tools detect external calls, but only if the code of the attack is provided as well. However, this approach might be effective for checking a possible attack, so the developers themselves act as malicious actors.

Manticore could bridge this gap by adopting two different running modes, so the user, knowing the limitation of each way of analysis, can combine those for obtaining a valuable result.

The strengths of this group of tools without specifications involve their speed, Slither was the fastest tool, installation and plug and play approach. Those just require the solidity file of the smart contracts. A user can use those for checking possible reentrancy risks and as the first step of the analysis.

6.4 Effective Analysis

In our work, we took into consideration the selected tools individually.

We run those per time focusing on the results of each one, and providing a comparison between those.

During an audit or a security report, a tester runs multiple of those for discovering vulnerabilities and bugs. A better way to fulfil this goal is using a combination of those. The tools without specification have, in our experience, an easier installation and usage, due to a reduced amount of external dependences and writing down specification is not required. Those can detect well know vulnerabilities and cover a predefined set of those.

Some of the tools with specifications we dealt with had some limitations regarding the external calls: just SolcVerify covered this set of vulnerabilities. For covering this limitation, a combination of tools would be a solution.

We should consider a tool without specification, Slither, and one with, Echidna. This combination has an effective result in terms of the speed of the analyses and amount of vulnerabilities covered. Slither has the role of detecting basic issues and reentrancy, on the other hand, Echidna can be used for the detection of a vulnerable implementation of the logic of the program. Since the grammar of this is similar to Manticore, an efficient solution would be to implement the same specification using it, which implements a different logic for scanning.

Formal verification resulted powerful for scanning possible problems, but SolcVerify and Celestial require to write down the specification for all the contracts for obtaining a consistent result. Certora has the strength of having implemented libraries which are mostly used in real-world cases (as OpenZeppelin ones,Aeve protocol). A facilitating aspect of this tool is the possibility to write down the specifications on just the properties we want to check and the possibility to code those in terms of function. It allows for the definition of specific preconditions, adding conditions to the environment. Another strenght of this tool is the possibility of using it as SaaS, consequently, the computational effort is demanded to another computer.

SolcVerify is the tool which obtained the best results in term of discovering vulnerabilities. One of the drawback we found is the è

A possible effective combination can include Echidna and Certora for covering the part of bugs in the logic and possible attacks, plus Slither for verifying the absence of possible reentrancy.

In term of effectivenss of discovering vulnerabilities, SolcVerify had the best behaviour, but it had issues during the installation phase and writing down the specification per each function increases depending the size of the smart contract. The presented combination is based on the facility during the writing the specifications and installation.

7 Conclusion

The security of smart contracts is a field which has been growing, as the popularity of blockchains and decentralised applications (web3, NFTs...) has been getting more spread.

This work presents the behaviour of analysis tools in a real-world context, providing insights into the process of definition of properties, usage of the tools and finding vulnerabilities. it is addressed to present a possible approach in real situations.

Our results do not deal with benchmarks but with attacks, and a comparison of the selected tools is produced. It deals with multiple aspects of those, starting with the installation and the user experience, involving the performance and effectiveness of the analysis.

The aim is to study the aspects of those in a situation as similar as possible to realty. The last chapters deal with their behaviour, understanding of which approach is more effective. Moreover, this work covers even the process of definition of the properties.

7.0.1 Critical reflection

This thesis aims to provide results dealing with real-world cases. The objectives of the analyses were smart contracts written in Solidity, which involves even EVM-compatible blockchain for having a wider view of the topic.

Security of smart contracts is fundamental for granting the correct behaviour of a decentralised application, but a limitation is a possibility of multiple threats involving the infrastructure of the system itself. Possible attacks on the blockchain are not covered by this field. The network itself could be vulnerable and a user could risk getting the private key of the wallet stolen.

The definition of the specifications is fundamental. This work involved attacks that happened in recent years. The focus was on the vulnerable parts, for understanding which vulnerabilities were exploited by the attackers. Dealing with the reviewed attacks, the way those were computed is known. The difficulties dealt with the definition of the properties and the correct usage of the tools. Reviewing a smart contract involves the additional task of trying to define all the possible vulnerabilities and writing those in form of specifications, for effective results from the tools providing custom analysis.

7.0.2 Outlooks

Further work can use the same tools, but analyse new attacks for evaluating the effectiveness of those tools in new real worlds cases.

It can provide the starting point for a code review, supporting developers and security analysts for the choice of tools for their experiments and tests, based on a given selection criteria.

A Appendix

A.1 First Appendix Section

Figure A.1: A figure

…