Security Analysis Tools for Ethereum 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

English abstract.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Blockchain represents one of the most popular trends in finance and computer science, during the last few years the number of investments has been growing exponentially. According CoinGeko [9], the crypto market's value is standing around $2 trillion.

Bitcoin can be considered the "father" of this technology. Nakamoto [27] depicted that in his paper, and in the early 2009, it was effectively launched and the cryptocurrency Bitcoin was introduced. CoinGeko [9] states the value of Bitcoin around $38,553.70 and its market capitalization more than $700 billions.

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. Buterin [6] 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 give the developers the opportunity 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, and many others.

Interest in such a market has grown even among malicious attackers. Attacks such as the "Parity Wallet Hack" and the "Decentralized Autonomous Organization Attack" cost millions of dollars simply because of naive bugs in the smart contract code. Blockchain and smart contract technologies have multiple aims, but unfortunately, new applications based on them still contain bugs and multiple vulnerabilities, which cause several issues for the end-users. Most of the use of this technology relates to finance or certifications, therefore integrity, authentication and authorisation in transactions are mandatory. The research field behind blockchain technology is growing, as well as the one concerning its security and accordingly, many analysis tools were developed. These incorporate various strategies for performing the analyses, concerning the technical aspects of smart contracts, so these would work differently according to the object of the analysis.

The topic that will be addressed in this thesis work is the analysis of smart contract security properties with the usage of tools. It involves the understanding of smart contracts properties and the comparison between different tools, providing insight regarding their behaviours in different contexts.

Table 1.1: Gantt Chart Tasks

Nunber	Name
T1	Research of real-world exploits and vulnerabilities
T2	Tools selection
T3	Installation and usage of tools
T4	Implementation of smart contracts
T5	Planning of the evaluation
T6	Testing smart contracts with tools
T7	Comparison of tools and properties
T8	Writing the thesis

1.2 Research Goals

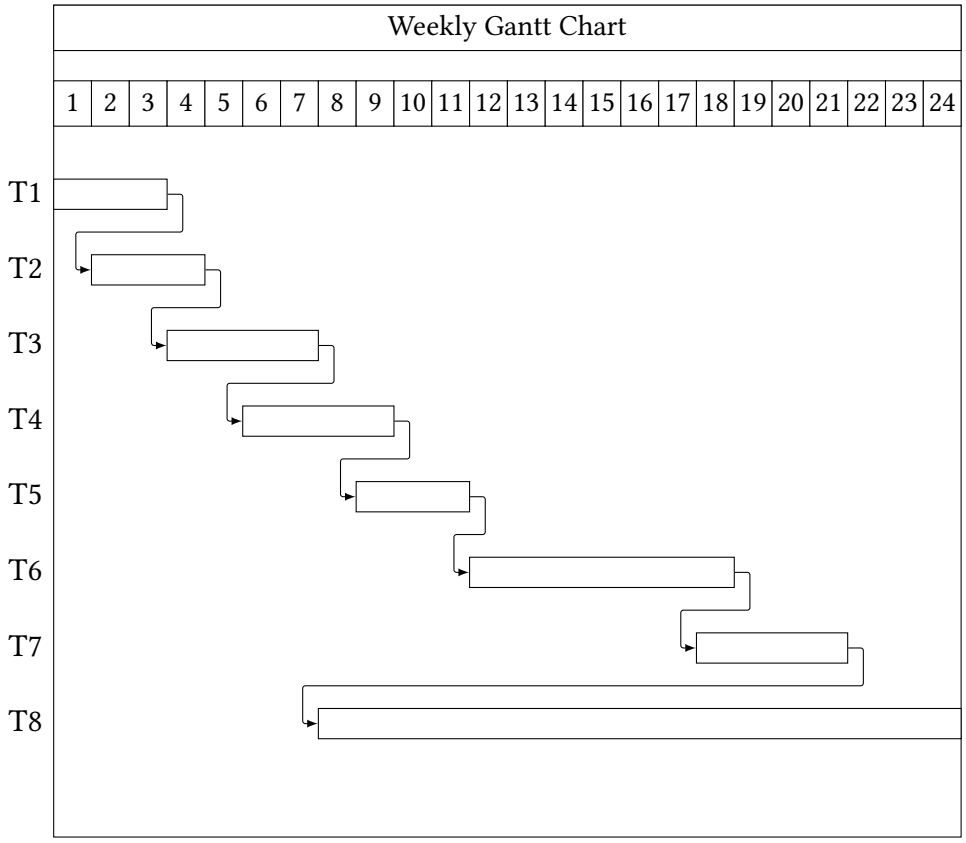
Research Question: How do state-of-the-art analysis tools for Ethereum/Solidity perform (on different classes of properties/bugs)?

This thesis focuses on a dozen analytic tools, which we choose based on the type of analysis, trying to have a range of different typologies. We will test them on vulnerable smart contracts and figure out which properties are violated during real-world exploits. Furthermore, we are going to compare the tools, based on their performance, in particular, the criteria for the evaluation can cover the completeness of the analysis, the amount of found vulnerabilities and the number of false positive and negative. The execution time is crucial too, we want to understand how long it takes for finding a vulnerability. The time for the configuration and the report interpretability are parameters for defining how much a tool is user friendly.

For answering the research question, we will give an answer to sub questions such as:

- How does a tool perform the analysis?
- Which properties are relevant for smart contract security?
- Which ones have been violated in real-world exploits?
- Which tools detect which class of vulnerabilities?

1.3 Research Approach



1.4 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 Tang et al. [40], Angelo and Salzer [3], Tolmach et al. [42], Sayeed, Marco-Gisbert, and Caira [31], Heidelind [19]. 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 (Feist, Grieco, and Groce [11], Mueller [26]) 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 (Certora [8], Hajdu and Jovanović [15], Dharanikota et al. [10]) and fuzzing (Grieco et al. [14]).

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 Preliminary Knowledge

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 the reading of this chapter for those who are not keen on these topics, otherwise, the reader can directly jump at chapter 3.

2.1 Blockchain

Sunyaev [38] provides the definition of Distributed ledger technology (DLT). This term has the role of umbrella term to encompasse multi-party systems that operate in an environment with no central authority, despite parties who may be unreliable or malicious.

Blockchain technology is a specific subset of the broader DLT universe that uses a particular data structure consisting of a chain of hash-linked blocks of data. It 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 that is constantly expanding. One of the characteristic is the possibility to generate a digital currency. A cryptocurrency can be defined as a medium of exchange secured by the ledger. A medium of exchange is anything widely accepted as payment for goods and services, and a ledger is a data store that keeps track of transactions. Most of the blockchains implemnt a cryptocurrency by designed, called "native token".

The term "blockchain" is form by "block", which refers to state and databeing 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. Due to the fact that any alteration to any historical block would render

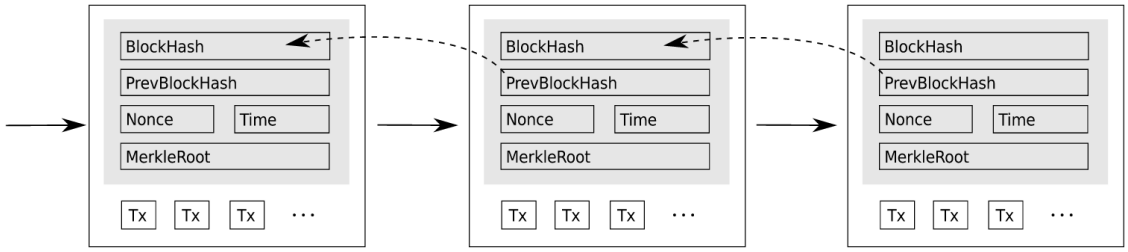


Figure 2.1: Simplified blockchain (image from [43]).

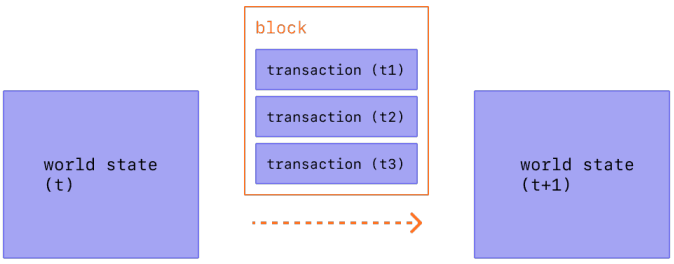


Figure 2.2: Change state overview(image from [13]).

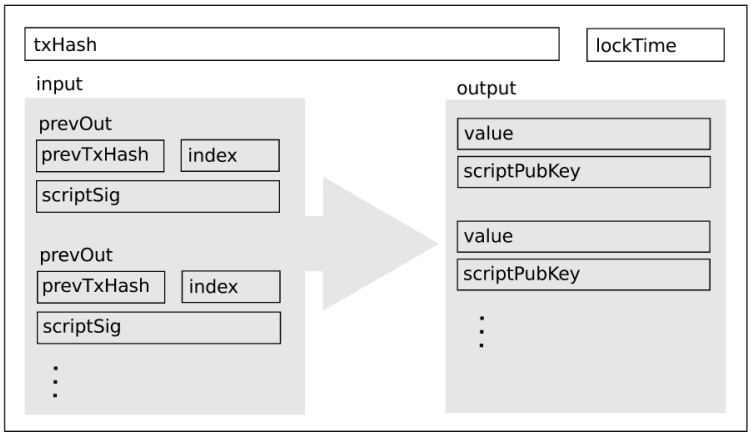


Figure 2.3: Trasnactions scheme (image from [43]).

all subsequent hashes incorrect and cause everyone using the blockchain to become aware of it, this prevents fraud.

Transactions are cryptographically signed instructions from accounts, which are the entities, having the possibility to get a balance and interact with the system. Accounts can be user-controlled or deployed as smart contracts.

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.

The entities, the computers, taking part of the network, are colled "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 is required for blockchains in order to implement this distributed agreement. It is foundamental for creating agreement within the network and defines which node is the the blocks are published in the network.

Following the survey of Tschorsch and Scheuermann [43], 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;
- anonimity, regarding the cryptographic aspects, such as the asymmetric key and the hashing of blockchain data;
- 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 Szabo [39]. 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.

2.2 Ethereum: a Smart Contract Platform

Ethereum is a blockchain that was developed by Buterin [6], presented in the 2014. A single, canonical computer (known as the Ethereum Virtual Machine, or EVM) exists in the Ethereum universe, and everyone on the Ethereum network agrees on its current state. Every Ethereum node, or participant in the network, maintains a copy of this computer's state.

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. The same procedures guarantee that every transaction is signed and carried out with the necessary "permissions."

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 it has a valid nonce. A miner who is frantically trying to make a block will continually run a dataset which can only be obtained by downloading and executing the entire chain—through a mathematical function. According to the block difficulty, the dataset is used to build a mixHash below a target nonce.

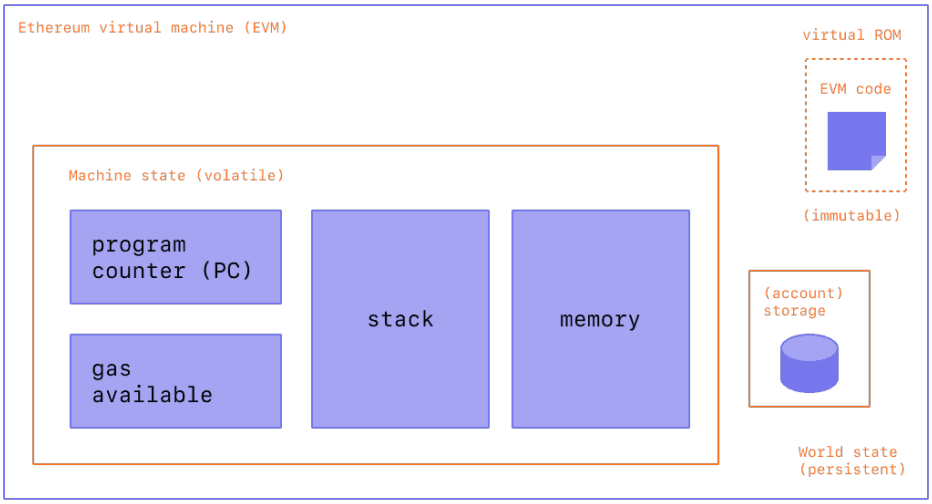


Figure 2.4: Diagram of EVM (image from [13])

In other words, the miners take part in this sort of race and the one who succedes to find the nonce, has the right to publish the block and get the rewards. Once generated, this is incredibly easy for other miners and clients to verify.

The rewards involve the native token of the Ethereum: ether (ETH).

Foundation [13] described its technology as a Smart Contract platform. Those are described as programs that run on the Ethereum blockchain. Their codes 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.

Those can be written in multple program languages defined by the Ethereum Foundation, which states that Solidity is the mostly used and mantained. 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.

The main limitation of those deals with their their inability to send HTTP requests, which is defined by desing. 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, which is vital for security and decentralisation.

—-> Eplain Some Vulnerabilities? Reentrancy and Fallback

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 to give uniqueness to digital data thanks the NFTs.

Decentralised Finance Multiple investors have been attrected by new possibilities offered by the "Decentralised Finance" (DeFi). An article on the European Central Bank website ([5]) deepenes this topic. The main strenght 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 the smart contracts. In most of the 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 is not any strong regulamentations and many investments have been involeved in DeFi, multiple malicious hackers used this for their benificious.

One of the target of the attackers is a particular type of decentralized exchange (DEX) named autonomous market maker (AMM). Academy [2] presentes it as a DEX based on a mathematical formula. Its peculiarity is how it estimates the assets price: according to a pricing algorithm. It is considered autonomous and decentralised because the user do 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 is 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 Academy [1], is the measure of how easily you can convert an asset into cash or another asset. In other words, it depicts how the convertion of an asset in another one affects the price of the asset. A 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.

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. For example, considerig a token named MiTHC and ETH, a pool is generated associating an amount of MiTHC with an amount of ETH. If the pool is generated by 1000 MiTHC and 1 ETH, in that moment MiTHC value is 0.001 ETH.

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 of 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. To fulfil this goal, multiple techniques have been adopted.

Fuzzing OWASP [28] 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.

[32] 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 Baldoni et al. [4] 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 strengh of symbolic execution is the simultaneously 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 Real world Exploits

In this part of thesis, we introduce the objectives of the analyses: the real world exploits.

Table 3.1 illustrates the attacks involve in our work, providing a brief description of those.

This thesis involves real-world exploits that have been occurred two years ago, so 2020. Our choice tries to cover different aspects and the trend of the attacks. 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 picked attacks deal with significant 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.

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

1. the explanation of the protocol;
2. the exploit;
3. the properties involved in the attack.

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, the depicted properties are involved in drawing the specification of the tools.

Table 3.1: Real World Exploits

Real world 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	Vulenrable depost withdraw process in a pool
XSURGE	Reentrancy in a token
BurgerSwap	Reentrancy in an AMM
DirtyDogs	Reentrancy in NFTs market

3.1 $34 Million stacks NFT Project Aku Dreams Smart Contract

Milmo [23] 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. They are made by a frontend part, but even by a backend one which relies on the blockchain.

Hassan [18] 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. Anyone who paid the higher amount would get a refund. Unfortunately, the launch was corrupted, since the errors in the codes made the project open to exploits. An attacker could block the withdrawals and refunds while attempting to highlight the vulnerabilities within the project.

3.1.1 The exploit

The first part of the exploit involved the function processRefunds Listing 3.1.

This has the aim to refund the bid of the user who took part in the auction.

The problem relies on the for loop in line 11. It loops on all over the users, who need 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 problem relies on the "require" in the loop. 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.

```
1   function processRefunds() external {
2     ...
3     for (uint256 i=_refundProgress; gasUsed < 5000000 && i < _bidIndex; i++) {
4         bids memory bidData = allBids[i];
5         if (bidData.finalProcess == 0) {
6           ...
7           if (refund > 0) {
8               (bool sent, ) = bidData.bidder.call{value: refund}("");
9               require(sent, "Failed to refund bidder");
10          }
11        }
12        ...
13        _refundProgress++;
14    }
15    refundProgress = _refundProgress;
16  }
```

Listing 3.1: Function for refunding the users.

The second part of the exploit is characterized by a bug in the logic, which could not allow the developer team to withdraw the project funds.

The function claimProjectFunds (Listing 3.2), 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 boolean condition, contained in the "require" in the first line of the function, is the heart of the problem. The "require" 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 variable totalBids is increased every time a bid is placed, regardless of the user who computed it, shown in Listing 3.3 . The user can call the function _bid, for placing a bid, with an arbitrary amount of bids, but the variable refundProgress is increased every time a user is refunded. Consequently, if a user bought more than one bid, the amount of refounded 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". We verify if

the contract can always refund all the users. The property involves the sum of refunded wallets may be equal to the number of the counter which loops on the map containing the data. It is a postcondition, so it is proven that all the users are refunded if the function does not revert, which means that the function does not consider the case of error.

The other property regards the function claimProjectFunds. In the beginning, some requirements have to be fulfilled before the owner can obtain the rewards. Our focus is on the comparison between the counter of the refunded users and the total amount of bids. In this case, we use proof by contradiction. We check if the processRefunds variable is always less than the totalBids. The property should be proofed if we consider that at least one user placed more than one bid.

Property 1 postcondition refered to function processRefunds():

$$_refundProgress <= _bidInd \tag{3.1}$$

Property 2 precondition referred claimProjectFunds() :

$$refundProgress >= totalBids \tag{3.2}$$

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 could steal from project around $ 4 million. The target of the attack was the smart contract "Blacksmith.sol", available on GitHub `https://github.com/CoverProtocol/cover-token-mining/blob/main/contracts/Blacksmith.sol`, its bug had the result to mint more rewards to the miner.

Sawinyh [30] interviewed the co-founder of the Cover Protocol. In his article, he answers some questions about his project, regarding its functionality and road map. It was an active protocol on the Ethereum blockchain; the developer deployed version 2, because of the attack. Cover Protocol 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 [29] 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.

```
1   function deposit(address _lpToken, uint256 _amount) external override {
2       ...
3       Pool memory pool = pools[_lpToken];
4       updatePool(_lpToken);
5
6       Miner storage miner = miners[_lpToken][msg.sender];
7       BonusToken memory bonusToken = bonusTokens[_lpToken];
8       _claimCoverRewards(pool, miner);
9       _claimBonus(bonusToken, miner);
10      // variable pool is not updated
11      miner.rewardWriteoff = miner.amount.mul(pool.accRewardsPerToken).div(CAL_MULTIPLIER)
    ;
12      ...
13  }
```

Listing 3.4: Deposit function.

Listing 3.4 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. The property relies on how it is computed. It is not estimated considering the correct parameters of the pool.

We compare the miner.rewardWriteoff with its recompute mathematical 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 post-condition involves the estimation of the reward inside the function deposit.

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

bZx Documentation [7] 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 powerful 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.

Unfortunately, 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 named LoanTokenLogicStandard. It 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.

Listing 3.5 shows the vulnerable function. The attacker called the function with the same amount of _from and _to. Since both addresses refer to the same one, line 27 decreases the balance of the address, but then line 31 increases the same balance. The problem relies on the estimating of the amount: it is the sum of the sent token and a variable, which stored the value of the balance before the transaction.

```
1   ...
2   function _internalTransferFrom (address _from, address _to, uint256 _value, uint256
    _allowanceAmount)
3       internal
4       returns (bool)
5   {
6       //The heart of the vulnerability
7       uint256 _balancesFrom = balances[_from];
8       uint256 _balancesTo = balances[_to];
9
10      require(_to != address(0), "15");
11
12      uint256 _balancesFromNew = _balancesFrom
13          .sub(_value, "16");
14      balances[_from] = _balancesFromNew;
15
16      uint256 _balancesToNew = _balancesTo
17          .add(_value);
18      balances[_to] = _balancesToNew;
19      ...
```

```
20    }
```

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 Listing 3.6 presents some differences. The operations regarding the receiver's balance are computed, then those which deal with the sender's one (16-20).

```
1     function _internalTransferFrom(address _from, address _to, uint256 _value, uint256
      _allowanceAmount)
2         internal
3         returns (bool)
4     {
5         ...
6         require(_to != address(0), "15");
7         uint256 _balancesFrom = balances[_from];
8         uint256 _balancesFromNew = _balancesFrom
9             .sub(_value, "16");
10        balances[_from] = _balancesFromNew;
11        uint256 _balancesTo = balances[_to];
12        uint256 _balancesToNew = _balancesTo
13            .add(_value);
14        balances[_to] = _balancesToNew;
15
16        return true;
17    }
```

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. We define 2 properties for defining the correct execution of the function. Both of those are broken by the wrong implemntation of the function.

The first one is a post-condition. It involves the correct estimation of the balance of the addresses involved in the operation, the parameters from and to. The balance of the sender should decrease and the one of the receiver should increase.

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

The other one is an invariant. It states the total sum of balances should be less than the variable total supply.

$$sum_of_uint(balances) <= totalSupply \tag{3.5}$$

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. For clarifying this, let's consider the following example. Bob can send (WBNB+SPARTA) into the WBNB-SPARTA pool and get Liquidity Pool (LP) tokens back, redeemable for the underlying assets.

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 *What Is a Flash Loan?* and *Understanding Flash Loans In DeFi* give a defintion flash loan.

Flash loan A flash loan 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.

3.4.1 The exploit

The exploit involved 2 contracts of protocol: Utils.sol and poolFactory.sol. The latter implements the strategy for the management of the liquidity in the pool and the former provides support functions. The mistake of the developers was not to consider the updated value of underlying assets. Those are stored into the variables (baseAmount, tokenAmount) and estimated with iBEP20(token).balanceOf(pool) and iBEP20(base).totalSupply().

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

```
1
2   function calcLiquidityShare(uint units, address token, address pool, address member)
    public view returns (uint share){
3       // share = amount * part/total
4       // address pool = getPool(token);
5       uint amount = iBEP20(token).balanceOf(pool);
6       uint totalSupply = iBEP20(pool).totalSupply();
7       return(amount.mul(units)).div(totalSupply);
8   }
```

Listing 3.7: calcLiquidityShare function

It should get the balance of the underlying asset in the pool (Listing 3.7). 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 variable (baseAmount, tokenAmount). The removeLiquidity() calls calcLiquidityShare on TOKEN and BASE(Listing 3.8). It fails to synchronize the balances of the underlying assets and the variables which store the amount of the assets.

```
1    // Remove Liquidity for a member
2    function removeLiquidityForMember(address member) public returns (uint outputBase, uint
     outputToken) {
3        uint units = balanceOf(member);
4        outputBase = iUTILS(_DAO().UTILS()).calcLiquidityShare(units, BASE, address(this),
     member);
5        outputToken = iUTILS(_DAO().UTILS()).calcLiquidityShare(units, TOKEN, address(this),
      member);
6        ...
7        return (outputBase, outputToken);
8    }
```

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 (baseAmount, tokenAmount). 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.

```
1    function _decrementPoolBalances(uint _baseAmount, uint _tokenAmount) internal  {
2
3        uint _removedBase = iUTILS(_DAO().UTILS()).calcShare(_baseAmount, baseAmount,
     baseAmountPooled);
4        uint _removedToken = iUTILS(_DAO().UTILS()).calcShare(_tokenAmount, tokenAmount,
     tokenAmountPooled);
5        baseAmountPooled = baseAmountPooled.sub(_removedBase);
6        tokenAmountPooled = tokenAmountPooled.sub(_removedToken);
7        //Wrong Estimation of these 2 variables
8        baseAmount = baseAmount.sub(_baseAmount);
9        tokenAmount = tokenAmount.sub(_tokenAmount);
10   }
```

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

A solution for this bug is shown in Listing 3.10. 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 miscalculates the value because it considers the balance of the account, which can be simply modified by transferring tokens into the pool. A proposed solution implies a function, synch, which synchronises the variables baseAmount and tokenAmount, which keep track of the underlying assets in the contract. Function synch should be called in the buggy function.

This one is encompassed in the definition of the property. It is called the end of the process of removing the liquidity for a member. The postcondition specifies that the function synch should not update the two variables.

$$nomodifiesbaseAmount, tokenAmount \tag{3.6}$$

Another way for checking the property would be checkig if the estimated returned value of the function calcLiquidityShare(...) uses the correct parameters.

$$\begin{aligned} &iBEP20(token).balanceOf(pool) == baseAmount \\ ||&iBEP20(token).balanceOf(pool) == tokenAmount \end{aligned} \tag{3.7}$$

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 Finance [12], 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

The article written by team, gets more in deep into the technical details involved in this exploit. The target of it was the contract MasterUranium, specifically the part 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 _amount input argument larger than "0".

The user.amountWithBonus increases by adding the _bonusAmount. The user.rewardDebt is calculated by the end of the function, with user.rewardDebt = user.amountWithBonus.mul(pool.accRadsP When the function returns, the both variables are greater than 0.

```
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 equal to and user.rewardDebt equal to 0. The fundamental variable user.amountWithBonus is still larger than 0. It is exploited during the last step.

```
    // 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. LThe 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. Checking the transcation on BSCscan, it is shown how many times the attacker replicated this methodology.

3.5.2 Properties

The logic of the smart contract vulnerability is contained in the procedure of estimation of users' rewards.

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 user struct, which keeps track of the amount and the bonus, just increases even if the user receives the reward and it is withdrawing.

For the detection of the vulnerability, we specify the property as a postcondition of the function withdraw.

$$\begin{aligned} &userInfo[msg.sender].bonus - userInfo[msg.sender].rewardDebt -> \\ &userInfo[msg.sender].bonus < old_value_of(userInfo[msg.sender].bonus) \end{aligned} \tag{3.8}$$

3.6 XSURGE on BSC Chain

The *xSurge Assets* [45]'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. *XSURGE on the BSC Chain was Attacked by Lightning Loans — A Full Analysis* [46] 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 (Referenceslst:SellSURGE), 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. value," use transfer or send to transfer.

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.

$$\_totalSupply <= old\_value\_of(\_totalSupply) \tag{3.9}$$

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

$$sum_of_uint(_balances) <=_t otalSupply \tag{3.10}$$

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 Leiba [21].

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, Listing 3.15. 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.

```
1
2   function withdrawFromBSC(bytes calldata _signature, bytes32 _paybackId, address _token,
    uint _amount) external payable {
3       require(executedMap[_paybackId] == false, "ALREADY_EXECUTED");
4
```

```
        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.

$$old\_value\_of(address(this).balance) == address(this).balance - \_amount \tag{3.11}$$

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 Listing 3.16. 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()]);
```

```
26      }
27
28      function getUserClaimableTicketCount(address user) public view returns (uint256) {
29          return presaleNumOfUser[user].add(publicNumOfUser[user]).sub(totalClaimed[user]);
30      }
31      ...
32      }
```

Listing 3.16: DirtyDogs NFT contract

3.8.2 Properties

The attacker could exploit the smart contract due to a vulnerability in the function claimDogs, 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(...).

$$sum_of_uint(totalClaimed) <= effclaimed \tag{3.12}$$

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. Table 4.1 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. We decided to discard the ones which have not been updated for more than two years. We decided to consider just open-source tools because their architecture descriptions and documentation are available. We made an exception in Certora, which is not open source. It has the premium version and the free one, we adopted the second one and the free provided documentation online.

Table 4.1: Tools Classification

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

The objectives of the analyses are smart contracts involved in real-world exploits, in most cases, the attackers forced a bug involving the logic of the software itself, more than a vulnerability regarding its grammar of it. For this reason, we mostly picked tools which could even have custom specifications.

4.1 Tools with multiple mood running

This section has the aim to describe the tools classified as "with multiple moods running". Even other tools have more moods 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 So, Hong, and Oh [35]. 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 (So et al. [37]). 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 Listing 4.1 shows an example of output of SmarTest, which provides the sequence of funtions for exploiting the found bug.

```
1  [5] [IO] line 39, (balance[_to] + _value) : disproven, 14.528264s
2  1: Example
3     {}
4     {msg.sender: #x0000000000000000000000000000000000010000,
5      msg.value: 0}
6  2: approve
7     {_spender: #x0000200000000000000000000000000000000000,
```

```
        _value:
    443657929256647019060809961937247473266455737933365557898023977251370916945 92}
       {msg.sender: #x0000000000000000001000000000000000000000,
        msg.value: 0}
    3: mintToken
       {_target: #x0000000000000000001000000000000000000000,
        _amount:
    873712858315893576366698616447642418058187921737390874086323388903712998031 36}
       {msg.sender: #x0000000000000000000000000000000000010000,
        msg.value: 0}
```

Listing 4.1: SmarTest Example Report.

The detection of the following six types of security-critical vulnerabilities are supported by the tool: integer over/underflow, assertion violation, division-by-zero, ERC20 standard violation, Ether-leaking vulnerability (e.g., unauthorized access to transfer), and suicidal vulnerability (e.g., unauthorized access to selfdestruct). 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

Mossberg et al. [24] describe in their paper an open-source dynamic symbolic execution framework called Manticore for analyzing binaries and Ethereum smart contracts.

The adaptable architecture of Manticore enables it to accommodate both common and uncommon execution contexts, and its API enables users to customize their analysis.

The keyword of this tool is dynamic symbolic execution, which is the implemented analysis technique. It examines a state space with a high level of semantic awareness. Dynamic symbolic execution 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 programme inputs necessary for the corresponding paths to be followed.

Figure 4.1 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.

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 traditional 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.

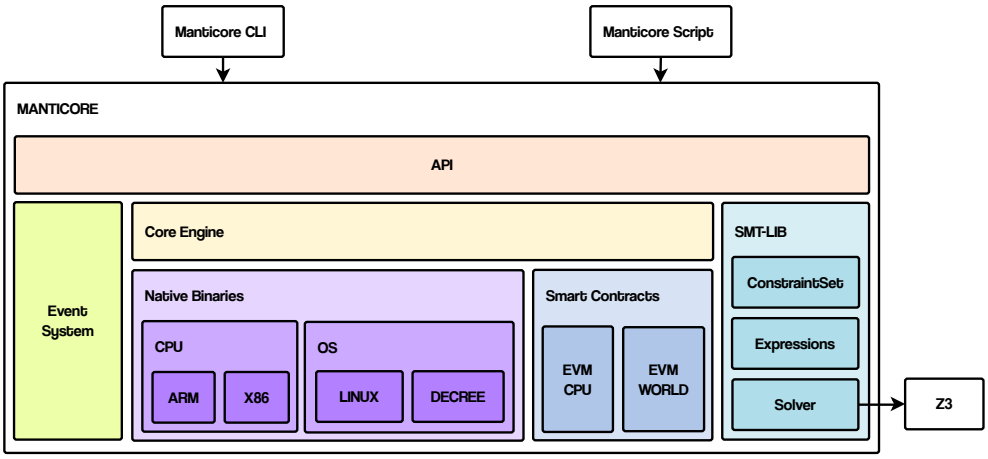


Figure 4.1: High-level architecture diagram (Figure from [24]).

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 one of the most well-known and used tools 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 *Certora Documentation*

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 result is the proof of the rule, otherwise, if the rule is disproved, the solver also provides a concrete test case demonstrating the violation.

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. For coding these, a similar language to Solidity is used. Multiple "require" definitions can be stated at the beginning, for expressing a condition to be realised before the running of it. Functions from the smart contract can be called. A requirement is that each rule has to conclude with assert, containing a boolean condition.

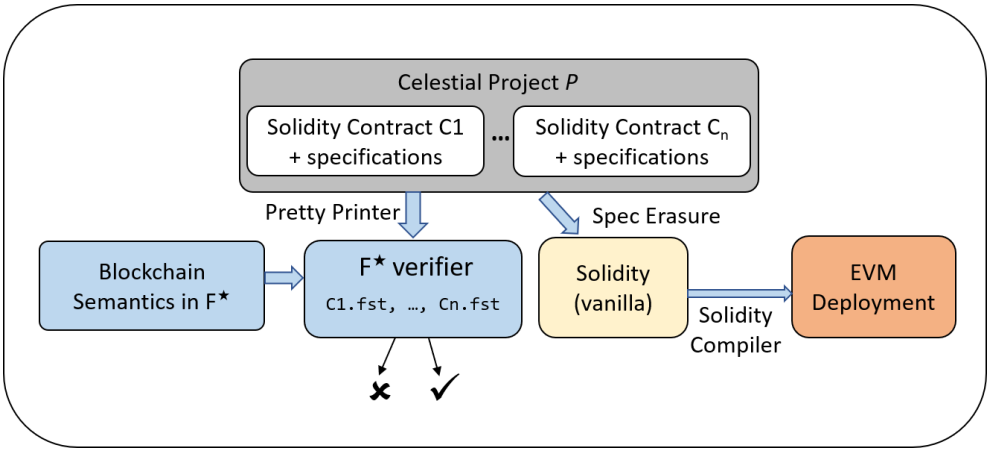


Figure 4.2: Celestial Architecture

Listing 4.2 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.

```

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);
    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. Figure 4.2 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.

Listing 4.3 shows an example of input file. The invariants are expressed in a sort of function. At the beginning of a function, the specification can be expressed, regarding precondition, postcondition and so on. One of these can involve the keyword modifies, which 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 gave the same reasons for involving F* for the formal proof in a blockchain context. First, it offers SMT-based automation, which is sufficient for the completely automated verification of real-world smart contracts. Second, F* 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 we only use its first-order subset with quantifiers and arithmetic, F* 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

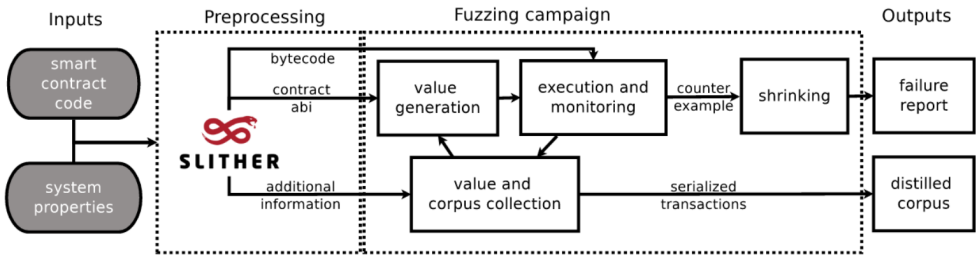


Figure 4.3: Echidna architecture (image from [14]).

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. Programmers can use the tool to create and support the specifications of their contracts without making any assumptions about the behaviour of external contracts.

4.2.3 Echidna

Echidna is an open-source smart contract fuzzer, developed by Grieco et al. [14], which makes it easy to automatically generate tests to detect violations in assertions and custom properties. Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities during fuzzing campaigns, Echidna supports three types of proper- ties:

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

Figure Figure 4.3 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 subsection 4.3.1, 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 Listing 4.4 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.

```
1 contract Token{
2     mapping(address => uint) public balances;
3     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

Hajdu and Jovanović [15] 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 Hajdu, Jovanović, and Ciocarlie [17] explain, Solc-verify translates the annotated contracts to the Boogie Intermediate Verification Language (IVL). The key idea of the translation is to encode state variables as global heaps and functions as procedures. 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.

Listing 4.5 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.

Hajdu and Jovanović [16] 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

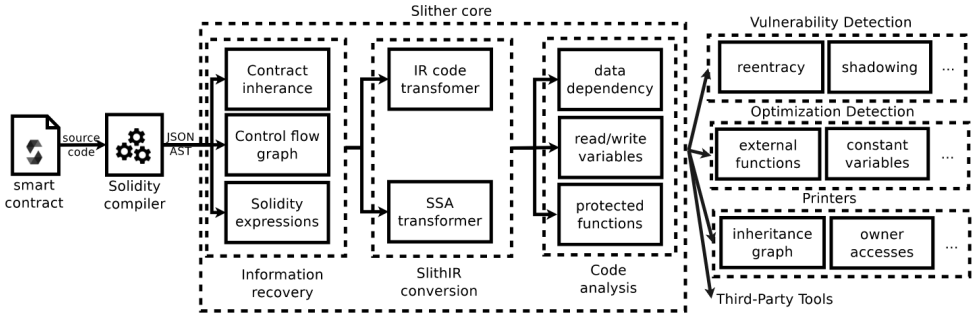


Figure 4.4: Slither architecture overview (image from [11]).

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 Feist, Grieco, and Groce [11] 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.
- 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.

Slither [33] 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 Mueller [26].

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. *SmartContractSecurity. SWC Registry* [34]'s taxonomy of vulnerabilities is used by Mythril to classify them. Listing 4.6 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 Evaluation

This chapter covers the outcomes of the tools, basically, what we obtained from our experiments.

Firstly, we describe the individual outcomes per tool, providing details about the running of those, problem that we faced off during the installation and usage. Then, we compare the tools based on these aspects, making subgroups due to their speifications.

5.1 Individual Outcomes per Tool

We describe the eventual problems during the installation and running phase, moreover, we show per each tool the outcomes of each analysis having as objective a specific attack. Per each tool, a table expresses the number of written lines of code considering the specifications, we did not count the ones which were copied from the target smart contracts. 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 ([25]) 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 Table 5.1 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 ([36]). The tool is built with OCalm, a program language, so we used for 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 Table 5.2 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.

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

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. 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.

Echidna

Certora

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	–	–	Found
BurgerSwap	–	–	Found
DirtyDogs	–	–	Found

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

SolcVerify

Mythril

Slither

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 usage and result of the tools. This section is struction in paragraphes, which deal with a specific aspecs.

Installation Table 5.7 collects the info about the installation detalais per each tool. We tested the tools in a linux eniveronoment, specifically Ubuntu 20.04, indeed we do not assure the proper functioning on other OSs.

The running mode involves the way a tool can be run, we classify a tool with multiple running mode if its grammar effectivly change 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 embedde the exteranal elements, so we do not count the amount of libraries of the same language. All the tools require Solc, the solidity compiler. The tools which gave us problem with the external dependences are Celestial and SolcVerify. The first one needs F* for running and we found many difficulties for its installation and for selecting its right version, based on the tool and the system environment. On the other hand, the second one involves the usage of Boogie, an intermediate verification language, which requires .NET, ([22]), which is an open source, cross-platform for building many kinds of applications mantained by Microsoft. Its intallation took days for configure the compatible version with the local system and the required by the tool.

Table 5.7: Installation

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

Since Certora operates in Service as a Service (SaaS) mode, it is the most compatible tool, it requires to be connect to internet.

Outcomes Table 5.8 has as rows the tools and as columns the real-world exploits. It shows which tool was able to scan the specified vulnerability involved in the attacks. The caption states the symbol we have used for the classification:

- ✓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.

We can state that the tools with specifications could not detect reentrancy in most of the cases. We classify that case as discarded. 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 could not analyse BZX exploit, because it involved the keywords "storage" and "memory", which cannot be scanned by the tool.

The tools without specifications gave warning in most cases, which can be used as a hint for scanning the vulnerabilities.

Slither could not detect the exploits involving the logic of the program, moreover, it gives multiple warnings, which the developer took into count. An example is BZX, which involves the estimation time comparison based on timestamp, but for the tool, a comperison timestamp based is always classified as dangerous.

Exploits with no reentrancy

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	✗	✗	✗	✗	✗	✓	✓	✓

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

Table 5.10: Weaknesses & Drawbacks

Tools	Weaknesses & Drawbacks
Manticore	Reentrancy is not dectected by properties property based execution, very slow
SmartTest	Reentrancy is not dectected, the analyses are slow
Celestial	External calls are not considered, keywords storage and memory are not recognized
Echidna	Reentrancy is not dectected, no assetion mode for solidity 8
Certora	Reentrancy is not dectected, not open-sources
SolcVerify	It just gives warning, it does not provide a list of transaction for breaking the given prope
Mythril	Just flat contracts are allowed
Slither	Scan is based on the grammar, great amount of false negative

Table 5.11: Strenghts

Tools	Strenghts
Manticore	One of the mode cover the properties breaking and the scanner one covers the reentrancy
SmartTest	It allows to set the specific vulenrability to look for
Celestial	Possibility to use different version of F*
Echidna	Possibility to run in multiple modes with different grammar (tests or assertions breaking)
Certora	Implements the library of Openzeppelin, SAS no installation needed
SolcVerify	Intuitive specification language based on Annotations, it detects reentrancy
Mythril	It deos not need specification, but still provide list of functions for breaking detected vuln
Slither	Easiest installation, fastest tool that we used

Tools behavior

5.3 Weaknesses and Strenghts

We present the

6 Discussion

This chapter deals with the discussion of the obtained results. We provide a comparison between the tools about their behaviour, including our point of view on user experience.

6.1 Tools with Specifications

The considered tools cover three main strategies for security analysis: fuzzing, symbolic execution and formal verification.

The following paragraphes deal with comparison within this subgroup we want to stess for having a deepen view of the comparison and a last sub chapter for a general overview.

Formal verification Formal verification is a very powerful security approach, intending to prove or unproved the given specification. This perfectly fits with our research goals, such as the detection of bugs or vulnerabilities in our real-world cases.

We involved three different tools, implementing this approach, for our purpose. Regarding the results, it is clear that these were executed as fast as or even faster than the other ones. 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 aspects 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; I found it very intuitive and fast to write down the specifications.

Celestial architecture encompasses two steps: the translation from celestial file to f* and then its verification. The python script converts the ".cel" in f*, used for the proof or unproof. The provided file included the smart contract's source code plus the expressed specification. These are statements placed at the beginning of a function, otherwise, it is possible to create a sort of function, containing boolean formula, which is called by different specifications function with different parameters, it is useful for expressing the same specification for different purpose. I consider it the one with more limitations, regarding solidity grammar and reentrancy, because it could not detect the reentrancy vunlenaribilities and the Cover protocol attack, because the keywords "storage" and "memory" are rejected.

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. The preconditions, in this case, are expressed using the Solidity keyword "require" in the rule.

One of its strengths is the possibility to define the specifications we want to prove, without necessarily defining all the specifics for the rest of the functions. On the other hand, with Celestial and SolcVerify, we provided the specifications for all the functions for letting the tools work properly. Those could not prove the given properties, without the specifications for all the code.

Different stategies, similar grammar We considered two tools with similar grammar but implemented different analysis approaches: Manticore and Echidna. In both of the cases, we provided functions containing a boolean formula, which the tools try to break. From the results, we noticed that Echidna run faster and it worked for all the cases, but Manticore could cover the reentrancy vulnerabilities thanks to its changing architecture.

We encountered a common aspect between the grammar definition of the specification between the two tools Echidna and SmarTest regarding their "assertion" mode. Both of those require the user to write assertions and then these try to verify it or return the list of functions for breaking the rule. From our results, it is clear Echidna could obtain higher number of positive outomces and in less time rather than SmartTest.

6.2 Customized and Non-specific Analyses

The objectives of our analyses were smart contracts involved in real-world exploits. The attackers exploited a specific bug or lack of security in the logic of programs.

The specifications allowed the user to express the requirements of the program. These provide a customized analysis and its accuracy is demanded by the developer as well. Our work involved smart contracts with well-known vulnerabilities, but the definition of the properties is an indispensable and complex step. The tools without specification implement vulnerabilities detectors, which are stated in their paper. They have well-defined limitations, but strengths as well. These detect preconfigured vulnerabilities, so a warning can be run even if we expect that indeed the number of false negatives can be relevant. An example is Slither, which for every comparison of block time stamp give a warning. These warnings

Our results stress the impossibility of the tools without specification for the detection of specific vulnerabilities, but they could correctly detect the reentrancy cases. On the other hand, the ones with specifications had problems in the reentrancy detection. The limitations of some of these involve the external calls. SolcVerify was the only tool which could provide the possibility of reentrancy detection. Echidna, as Certora, developers teams specified the tools can detect the reentrancy, in the case an attack is provided, but we think this approach can be useful for checking a possible attack rather than detection of vulnerabilities.

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

6.3 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. 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. For covering this limitation, a combination of tools would be a solution. As an example, we 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 idea would be to implement the test using a tool, which implements a different logic for scanning. Formal verification resulted powerful for scanning possible problems, but Solc-Verify and Celestial need to write down the specification for all the contracts for obtaining a consistent result. Certora, which is not open source, has the strength of having implemented libraries which are mostly used in real-world cases (as OpenZeppelin ones). 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.

We suggest a combination of Echidna and Certora for covering the part of bugs in the logic and possible attacks, plus Slither for verifying the absence of possible reentrancy. We selected this combination based on the facility we had during the write down of the specifications and installation of those.

7 Conclusion

General overview to the research questios, new approches like Machine Learning, citining testing environment as Harhat, future of blockchain and adoption of solidity, different vulnerabilities regarding even the architecture itself. 1 or 2 pages

- How does a tool perform the analysis?
- Which properties are relevant for smart contract security?
- Which ones have been violated in real-world exploits?
- Which tools detect which class of vulnerabilities?

7.0.1 Critical reflection

Our thoughs and limitation on our work.
More language, vulnerabilities regarding Blockchain itself sybil attacks, eclipse attacks

7.0.2 Outlook for further research and development

Machine Learning, unit testing with Hardhat

Bibliography

[1] Binance Academy. *Liquidity Explained*. https://academy.binance.com. 2022. URL: https://academy.binance.com/en/articles/liquidity-explained.

[2] Binance Academy. *What Is an Automated Market Maker (AMM)?* 2020. URL: https://academy.binance.com/en/articles/what-is-an-automated-market-maker-amm.

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

[4] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

[5] Alexandra Born. *Decentralised finance: a new unregulated non-bank system?* URL: https://www.ecb.europa.eu/pub/financial-stability/macroprudential-bulletin/focus/2022/html/ecb.mpbu202207_focus1.en.html.

[6] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[7] *bZx Documentation*. https://docs.bzx.network/. Accessed: 2022-04-20. 2020.

[8] Certora. *Certora Documentation*. 2022. URL: https://docs.certora.com/en/latest/index.html.

[9] CoinGeko. *CoinGeko Web Site*. 2020. URL: https://www.coingecko.com/.

[10] Samvid Dharanikota et al. *Celestial: A Smart Contracts Verification Framework*. Tech. rep. MSR-TR-2020-43. Microsoft, Dec. 2020. URL: https://www.microsoft.com/en-us/research/publication/celestial-a-smart-contracts-verification-framework/.

[11] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts". In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, May 2019. DOI: 10.1109/wetseb.2019.00008. URL: https://doi.org/10.1109%2Fwetseb.2019.00008.

[12] Uranium Finance. *Uranium : post-mortem, v2, compensations*. 2021. URL: https://uraniumfinance.medium.com/uranium-post-mortem-v2-compensations-aac4b0706d7d.

[13] Ethereum Foundation. *Ethereum development documentation*. 2022. URL: https://ethereum.org/en/developers/docs/.

[14] Gustavo Grieco et al. "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 557–560. ISBN: 9781450380089. DOI: 10.1145/3395363.3404366. URL: https://doi.org/10.1145/3395363.3404366.

[15] Á kos Hajdu and Dejan Jovanović. "solc-verify: A Modular Verifier for Solidity Smart Contracts". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 161–179. DOI: 10.1007/978-3-030-41600-3_11. URL: https://doi.org/10.1007%2F978-3-030-41600-3_11.

[16] Á kos Hajdu and Dejan Jovanović. *SolcVerify*. https://github.com/SRI-CSL/solidity/blob/0.7/SOLC-VERIFY-README.md. 2021.

[17] Ákos Hajdu, Dejan Jovanović, and Gabriela Ciocarlie. *Formal Specification and Verification of Solidity Contracts with Events*. May 2020.

[18] Saeed Hassan. *NFT Project Aku Dreams Loses $34 Million To Smart Contract Flaw*. 2022. URL: https://bitcoinist.com/nft-project-aku-dreams-loses-34-million-to-smart-contract-flaw/.

[19] Rameder Heidelind. "Systematic Review of Ethereum Smart Contract Security Vulnerabilities, Analysis Methods and Tools". PhD thesis. Technische Universität Wien, 2021, p. 155.

[20] Alyssa Hertig. *What Is a Flash Loan?* 2022. URL: https://www.coindesk.com/learn/2021/02/17/what-is-a-flash-loan/.

[21] Oded Leiba. *Reentering the Reentrancy Bug: Disclosing BurgerSwap's Vulnerability*. 2020. URL: https://zengo.com/burgerswap-vulnerability/.

[22] Microft. *DotNet Documentation*. 2020. URL: https://docs.microsoft.com/en-us/dotnet/core/introduction.

[23] Dan Milmo. *NFTs market hits 22billions dollars as craze turns digital images into assets*. The Guardian online. 2021. URL: https://www.theguardian.com/technology/2021/dec/16/nfts-market-hits-22bn-as-craze-turns-digital-images-into-assets?CMP=Share_iOSApp_Other.

[24] Mark Mossberg et al. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. DOI: 10.48550/ARXIV.1907.03890. URL: https://arxiv.org/abs/1907.03890.

[25] Mark Mossberg et al. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. Nov. 2019. DOI: 10.1109/ASE.2019.00133. URL: https://github.com/trailofbits/manticore.

[26] Bernhard Mueller. "Smashing ethereum smart contracts for fun and real profit". In: Amsterdam, Netherlands: In 9th Annual HITB Security Conference (HITBSecConf), 2018. URL: https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf.

[27] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[28] OWASP. *Fuzzing*. URL: https://owasp.org/www-community/Fuzzing.

[29] Cover Protocol. *12/28 Post-Mortem*. 2020. URL: https://coverprotocol.medium.com/12-28-post-mortem-34c5f9f718d4.

[30] Nick Sawinyh. *Cover Protocol - Decentralized Insurance Marketplace*. 2021. URL: https://defiprime.com/cover-protocol.

[31] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. "Smart Contract: Attacks and Protections". In: *IEEE Access* 8 (2020), pp. 24416–24427. DOI: 10.1109/ACCESS.2020.2970495.

[32] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. "Chapter 2 - Basic formal verification algorithms". In: *Formal Verification*. Ed. by Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. Boston: Morgan Kaufmann, 2015, pp. 23–47. ISBN: 978-0-12-800727-3. DOI: https://doi.org/10.1016/B978-0-12-800727-3.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780128007273000022.

[33] *Slither*. https://github.com/crytic/slither. 2019.

[34] *SmartContractSecurity. SWC Registry*. https://swcregistry.io/. Accessed: 2022-04-15. 2020.

[35] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. "SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1361–1378. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/so.

[36] Sunbeom So et al. *VeriSmart GitHub*. 2021. URL: https://github.com/kupl/VeriSmart-public.

[37] Sunbeom So et al. *VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts*. 2019. DOI: 10.48550/ARXIV.1908.11227. URL: https://arxiv.org/abs/1908.11227.

[38] Ali Sunyaev. "Distributed Ledger Technology". In: *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Cham: Springer International Publishing, 2020, pp. 265–299. ISBN: 978-3-030-34957-8. DOI: 10.1007/978-3-030-34957-8_9. URL: https://doi.org/10.1007/978-3-030-34957-8_9.

[39] Nick Szabo. *Smart Contracts*. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[40] Xiangyan Tang et al. "The Vulnerabilities in Smart Contracts: A Survey". In: *Advances in Artificial Intelligence and Security*. Ed. by Xingming Sun et al. Cham: Springer International Publishing, 2021, pp. 177–190. ISBN: 978-3-030-78621-2.

[41] Certik team. *Uranium Finance Exploit Analysis*. 2021. URL: https://medium.com/shentu-foundation/uranium-finance-exploit-analysis-d135055d6a6a.

[42] Palina Tolmach et al. *A Survey of Smart Contract Formal Specification and Verification*. 2020. DOI: 10.48550/ARXIV.2008.02712. URL: https://arxiv.org/abs/2008.02712.

[43] Florian Tschorsch and Björn Scheuermann. "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies". In: *IEEE Communications Surveys and Tutorials* 18.3 (2016), pp. 2084–2123. DOI: 10.1109/COMST.2016.2535718.

[44] Dennis Van der Vecht. *Understanding Flash Loans In DeFi*. 2022. URL: https://10clouds.com/blog/defi/understanding-flash-loans-in-defi/.

[45] *xSurge Assets*. https://xsurge.net/surge-assets. Accessed: 2022-04-15. 2021.

[46] *XSURGE on the BSC Chain was Attacked by Lightning Loans — A Full Analysis*. https://beosin.medium.com/a-sweet-blow-fb0a5e08657d. Accessed: 2022-04-15. 2021.

A Appendix

A.1 First Appendix Section

Figure A.1: A figure

...