

# Secure Analysis Chain of Smart Contracts

Analyzing tools for vulnerabilities in Ethereum Smart Contract

Kavya Goyal

*Computer Science and Engineering*  
*Indian Institute of Technology Patna*  
Patna, India  
kavya\_1901cs30@iitp.ac.in

Priyanka Sachan

*Computer Science and Engineering*  
*Indian Institute of Technology Patna*  
Patna, India  
priyanka\_1901cs30@iitp.ac.in

**Abstract**—Smart contracts are computer programs that are hosted and executed on a blockchain network, written in Turing-complete languages like Solidity. They have capabilities of controlling large amounts of data over immutable deployed contracts. While creating a large system of decentralized applications has many advantages, it presents the potential for attackers to exploit the vulnerabilities of smart contracts. Over the years, tools have been developed for automating the analysis and testing of the vulnerabilities of smart-contracts. In this paper, we have aimed to analyse few of these tools namely Slither, Manticore and Mythril to evaluate the smart-contracts and the security vulnerabilities they propose. We have defined the research on the basis of four criterion of analysis namely static analysis, symbolic execution or dynamic analysis, fuzzy testing and formal verification. In this paper, we present the results of analysis of 143 smart contracts which have vulnerabilities listed under DASP (Decentralized Application Security Protocol) **Keywords:** smart-contract-security, smart-contract-vulnerabilities, smart-contract-tools, slither, mythril, DASP, SWC-Contracts

## I. INTRODUCTION

Over the years, blockchain technology has gained considerable attention because of its potential to move towards a more democratic, decentralized and scalable system. Among this, Ethereum Smart Contracts written in solidity have emerged and gained extreme popularity because of the extensibility offered of creating and writing decentralized applications or Dapp without intervention of third-party trusted authorities. Smart contracts are a powerful infrastructure for automation because they are not controlled by a central administrator and are not vulnerable to single points of attack by malicious entities. When applied to multi-party digital agreements, smart contract applications can reduce counter-party risk, increase efficiency, lower costs, and provide new levels of transparency into processes.

Writing secure smart contracts is a recurring challenge and to mitigate this many efforts have been made by research community to develop automated analysis tools that locate and eliminate vulnerabilities in smart contracts. Repeated security concerns have shaken the trust in handling billions of USD by smart contracts.

- 150M were stolen from the popular DAO contract in June 2016
- 30M were stolen from the widely-used Parity multi-signature wallet in July 2017

- 280M were frozen due to a bug in the very same wallet

Some of the reasons why smart contracts are particularly susceptible and vulnerable to attacks are 1) Due vulnerabilities in current programming languages used like solidity 2) Absence of adequate tools covering analysis and detection for all the attacks and vulnerabilities and 3) Immutability of already deployed smart contracts. As a result, smart contracts are susceptible to security vulnerabilities and malicious attacks, which further highlights the importance of smart contract security protection

The aim of this paper is to introduce some smart contract vulnerability analysis tools and analyze how they are able to detect the standard vulnerabilities listed under Decentralized Security Application Protocol (DASP) and analyse the smart contracts based on them. For the purpose of this paper, we will be working with some sample contracts. The result files include the analysis of 143 smart-contracts taken from various sources known to have DASP security vulnerabilities. We will be covering the analysis under the static, dynamic and formal-verification domains.

The rest of this paper is divided in the following way. Section II introduces the smart contract bug classes and the dataset. Section III introduces the domains of analysis. Section IV documents the analysis and results of sample contracts. Section V introduces the comparisons and conclusions

## II. SMART CONTRACTS AND BUG CLASSES

### A. Collection of Smart-Contracts

The dataset constructed used a set of solidity smart contracts that have known vulnerabilities. They have been extracted majorly from the Smart Contract Weakness Classification and Test Cases Registry. The contracts are annotated based on the DASP categories as introduced as below. The python notebook *Analysis-SWC-Contracts* can be run to extract and generate the smart contracts as seen in *Figure 1*. A total of 143 smart-contracts were downloaded in this way

### B. DASP Vulnerabilities and Bug Classes

Each of the smart contract is labelled under one of the 10 categories as show in *Figure 2*. Majority vulnerabilities of the smart contracts can be coupled and grouped under these categories

```

folder_path = 'downloaded_files/dataset'
number_of_contracts = 0
contract_mapping = {}
contract_paths = []

for control in os.listdir(folder_path):
    control_folder_path = os.path.join(folder_path, control)

    for vulnerable_contract in os.listdir(control_folder_path):
        contract_path = os.path.join(control_folder_path, vulnerable_contract)
        contract_paths.append(contract_path)
        number_of_contracts += 1
        contract_mapping[vulnerable_contract] = control

print('Number of contracts: ' + str(number_of_contracts))

```

Number of contracts: 143

Fig. 1. Extracting the dataset

Category	Description
Access Control	Failure to use function modifiers or use of tx.origin
Arithmetic	Integer over/underflows
Bad Randomness	Malicious miner biases the outcome
Denial of service	The contract is overwhelmed with time-consuming computations
Front running	Two dependent transactions that invoke the same contract are included in one block
Reentrancy	Reentrant function calls make a contract to behave in an unexpected way
Short addresses	EVM itself accepts incorrectly padded arguments
Time manipulation	The timestamp of the block is manipulated by the miner
Unchecked low level calls	call(), callcode(), delegatecall() or send() fails and it is not checked
Unknown Unknowns	Vulnerabilities not identified in DASP 10

Fig. 2. DASP categories and their description

- **Reentrancy (SWC-107)**

Reentrancy is an attack that can occur when a bug in a contract may allow a malicious contract to reenter the contract unexpectedly during execution of the original function. This can be used to drain the ether supplied as gas and lead to infinite usage of gas. Reentrancy vulnerability when exploited resulted in the famous DAO Attack causing losses of over 60 million.

- **Integer-Overflow-and-Underflow (SWC-101)**

The different *uintx* in solidity have different limits given as (uint8 : 255), (uint16 : 65535), (uint24 : 16777215) and so on. Overflows and underflows can occur and can be exploited as bugs giving incorrect results compromising the security policies.

- **Insufficient Access Control (SWC-105)**

One of the famous attacks, the *Parity Wallet Attack* occurred due to missing or insufficient access controls, by allowing malicious parties to withdraw some or all Ether from the contract account. This bug majorly occurs when a function is defined as a constructor and then holds the capability to be access by anyone to initialize or use it.

- **Insufficient Gas Griefing (SWC-126)**

Insufficient gas griefing can be done on contracts which accept data and use it in a sub-call on another contract. This method is often used in multi signature wallets as well as transaction relayers. If the sub-call fails, either the

whole transaction is reverted, or execution is continued.

- **DoS with (Unexpected) revert (SWC-113)**

A DoS (Denial of Service) may be caused when logic is unable to be executed as a result of an unexpected revert. This can happen for a number of reasons and it's important to consider all the ways in which your logic may revert.

- **Unprotected *SELFDESTRUCT* Instruction (SWC-106)**

One of the major functionality of smart contract is to transfer and receive ethers to and from other contracts. The *delegateCall* might be used to perform this transfer to

external contract. However, if the called external transfer function has self-destruction operations such as Self-destruct and Suicide, ether on the current contract (which calls this transfer function) is likely to be frozen due to such a self-destruction operation executed

- **Signature Replay Attacks (SWC 121)**

The utility might call to perform signature verification to improve extensibility and reduce wastage of gas. Signature Replay Attacks occur when malicious users use same hashes by replaying another users signature multiple times.

- **Transaction Origin Use (SWC 115)**

Smart contracts made over etheruem have a global *tx.origin* global variable which can be used to backtrack the entire call stack and return the original base address of the contract that has been called. This variable, if used for authorization and authentication, exposes the threat for attackers to use the fallback functions and steal Ether.

- **Unsafe Low Level Calls/Unchecked call return value (SWC 104)**

If the return value of a low-level call is not checked, the execution may resume even if the function call throws an error. This can lead to unexpected behavior and break the program logic. A failed call can even be caused by an attacker, who may be able to further exploit the application.

- **Block-Dependency-Attack** (SWC-114)

Transactions on Ethereum are logged in blocks and processed at some intervals. It takes about 10 minutes to mine a single block from the pool of blocks to be mined (also known as mempool). The fact that mempool is public gives rise to the possibility of malicious users to supersede by placing their own transaction executing the same, or a similar, action with a higher gas price.

- **Timestamp Dependence** (SWC-116)

The timestamp of a block, accessed by `block.timestamp` or `alias now` can be manipulated by a miner.

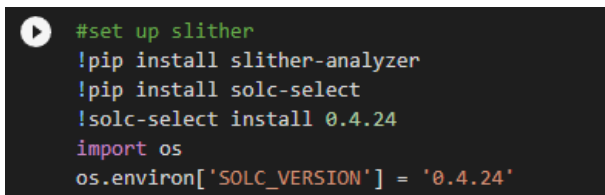
### III. DOMAINS OF ANALYSIS AND TOOLS

#### A. Static-Analysis

Static code analysis is a method of debugging by examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules. Static analysis identifies defects before we run a program. A static analyzer takes as input the source code of a smart contract and outputs results declaring whether a contract satisfies a property or not. It lists out the vulnerabilities detected on the contracts.

- **Slither:** Slither is a static analysis framework for Ethereum smart contract analysis to provide rich information, which combines data flow analysis and taint analysis. It aids in generating the intermediate representation named SlithIR, which employs a static single allocation (SSA) form and reduced instruction set to simplify the analysis process while retaining the lost semantic information when the source code is converted to EVM bytecode.

All of our 143 smart contracts were run and their result was saved in `143-SWC-slither-analysis-result.json` file where each entry has a key which is essentially of the format (category)(vulnerability) and the value is the result on analysis of Slither



```
#set up slither
!pip install slither-analyzer
!pip install solc-select
!solc-select install 0.4.24
import os
os.environ['SOLC_VERSION'] = '0.4.24'
```

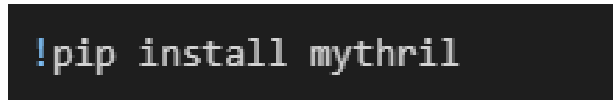
Fig. 3. Setting Up Slither

#### B. Dynamic Analysis

Dynamic analysis generates symbolic inputs (e.g., in symbolic execution or concrete inputs (e.g., in fuzzing) to a smart contracts function to see if any execution trace(s) violates specific properties. In other words, all the possible paths or states are explored to find the vulnerabilities. This form of property-based testing differs from unit tests in that test cases cover multiple scenarios and a program handles the generation of test cases.

1) *Symbolic Execution:* Symbolic execution is a method of executing a program abstractly so that it covers multiple execution paths through the code. The execution makes use of “symbols” as inputs to the program and the results are expressed in terms of the symbolic inputs.

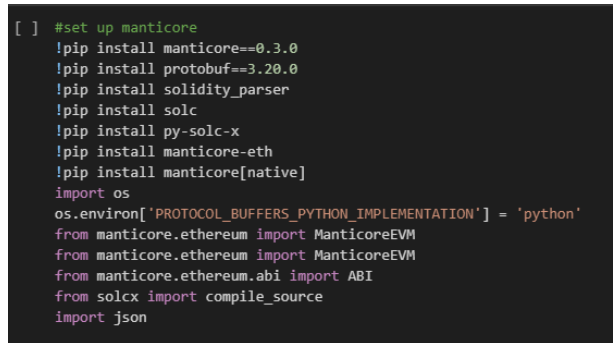
- **Mythril:** Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum, Hedera, Quorum, VeChain, Rookstock, Tron and other EVM-compatible blockchains. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. It's also used (in combination with other tools and techniques) in the MythX security analysis platform



```
!pip install mythril
```

Fig. 4. Setting Up Mythril

- **Manticore:** Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. It can execute a program with symbolic inputs and explore all the possible states it can reach by automatically producing concrete inputs that result in a given program state



```
[ ] #set up manticore
!pip install manticore==0.3.0
!pip install protobuf==3.20.0
!pip install solidity_parser
!pip install solc
!pip install py-solc-x
!pip install manticore-eth
!pip install manticore[native]
import os
os.environ['PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION'] = 'python'
from manticore.ethereum import ManticoreEVM
from manticore.ethereum import ManticoreEVM
from manticore.ethereum.abi import ABI
from solcx import compile_source
import json
```

Fig. 5. Setting Up Manticore

2) *Fuzzy Testing:* The process of Fuzzing involves throwing invalid, unexpected, or random data as inputs at a computer. Fuzzers repeat this process and monitor the environment until they detect a vulnerability. The purpose of fuzzing relies on the assumption that there are bugs within every program, which are waiting to be discovered. Therefore, a systematic approach should find them sooner or later.

- **Echidna:** Echidna is an open-source Ethereum smart contract fuzzer. Rather than relying on a fixed set of predefined bug oracles to detect vulnerabilities during fuzzing campaigns, Echidna supports (1) user-defined properties (2) assertion checking, and (3) gas use estimation.

#### C. Formal Verification

Formal verification can formally prove that a smart contract satisfies requirements for an infinite range of executions

```
sachan@CLOUD-DESK:~/Documents/Sem-8/Computer Security/smart contract analysis chain
/echidna-docker$ docker pull trallofbits/eth-security-toolbox
docker run -it -v "$PWD":/home/training trallofbits/eth-security-toolbox
```

Fig. 6. Setting up Echidna

without running the contract at all. With this, the question of verifying if a contract's business logic satisfies requirements is a mathematical proposition that can be proved or disproved. By formally proving a proposition, we can verify an infinite number of test cases with a finite number of steps.

- **KEVM Framework:** KEVM is a formal analysis framework, which utilizes the K framework to construct an executable formal specification based on the EVM bytecode stack. Further, KEVM serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM.

#### IV. ANALYSING ON SMART CONTRACTS

##### A. The Vulnerable/Attacker Contract

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.9;
```

```
contract VulnerableContract {

    uint256 public myVariable;
    uint256 public unusedVariable;

    constructor() public {
        myVariable = 123;
    }

    function updateVariable(uint256
        _newVariable) public {
        myVariable = _newVariable;
    }

    function uselessFunction() public {
        // This function has no effect
    }

    function setUnusedVariable(uint256
        _unusedVariable) public {
        unusedVariable = _unusedVariable;
    }

    function getUnusedVariable() public
        view returns (uint256) {
        return unusedVariable;
    }
}

contract AttackerContract {
    VulnerableContract vulnerableContract
    ;
```

```
constructor(VulnerableContract
    _vulnerableContract) public {
    vulnerableContract =
        _vulnerableContract;
}

function attack() public {
    vulnerableContract.updateVariable
        (0);
    vulnerableContract.
        setUnusedVariable(0);
    vulnerableContract.myVariable;
}
}
```

The vulnerabilities explored are:

- *Code with no effect:* The uselessFunction() function has no effect.
- *Presence of unused variables:* The unusedVariable variable is unused in the smart contract.
- *Incorrect Inheritance Order:* It is important to place the parent contract above the child contract in the contract definition. There is no inherent inheritance order in this smart contract
- *Outdated Compiler Version:* The smart contract uses an outdated compiler version. It is imperative that we use the LTS version of compiler without deprecations to ensure compatibility
- *Function Default Visibility:* The VulnerableContract constructor should have been declared as constructor() instead of function VulnerableContract(), since the latter has default visibility and could potentially allow unauthorised contract deployments. Need to use the appropriate visibility modifier for variables and functions..
- *Unchecked Return Values:* When a smart contract calls another contract, it expects the called contract to return a value of a certain type. However, if the called contract doesn't return the expected value, it can cause issues in the calling contract. Need to check the return values of functions that modify state variables.
- *Running Slither*

Fig. 7. slither on test-1.sol

##### • Running Mythril

Multiple vulnerabilities in the domain of

- ===== External Call To User-Supplied Address =====
- ===== State access after external call =====
- ===== Multiple Calls in a Single Transaction =====

were exploited and found using the tool. The result is also attached as *mythril-test-type1.txt*

### B. The safeMath Contract

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.9;

contract SafeMath {
    function safeAdd(uint a, uint b)
        public pure returns (uint c) {
        c = a + b;
        require(c >= a);
    }
    function safeSub(uint a, uint b)
        public pure returns (uint c) {
        require(b <= a);
        c = a - b;
    }
    function safeMul(uint a, uint b)
        public pure returns (uint c) {
        c = a * b;
        require(a == 0 || c / a == b);
    }
    function safeDiv(uint a, uint b)
        public pure returns (uint c) {
        require(b > 0);
        c = a / b;
    }
}
```

#### • Running Slither

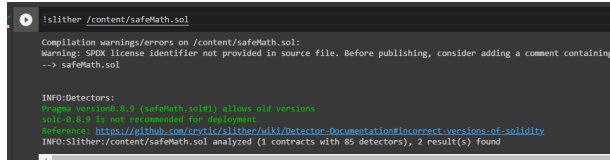


Fig. 8. slither on safeMath.sol

#### • Running Mythril

The following result was generated by Mythril on safeMath.sol

*The analysis was completed successfully. No issues were detected.* The result is also attached as *mythril-safeMath.txt*

#### • Running Manticore

Many states were generated. Some resulted in *nRETURN* and the other resulted in *REVERT* showing the possible executions. The result of all the states hence formed is attached in the *mc-core-sm49-f4-safeMath* folder

### C. The token/test-token contract

The *token.sol* is given by

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity ^0.5.0;
```

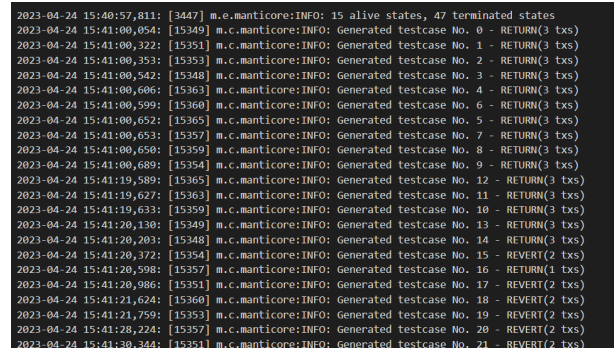


Fig. 9. manticore on safeMath

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

    function airdrop() public {
        balances[msg.sender] = 1000;
    }

    function consume() public {
        require(balances[msg.sender] > 0);
        ;
        balances[msg.sender] -= 1;
    }

    function backdoor() public {
        balances[msg.sender] += 1;
    }
}
```

Some logical tests were written to run fuzzy testing tool Echidna on it. The *testToken.sol* is given by

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity ^0.5.0;
```

```
import "./token.sol";
```

```
contract TestToken is Token {
    function echidna_balance_under_1000()
        public view returns (bool) {
        return balances[msg.sender] <=
            1000;
    }
}
```

#### • Running Echidna

Echidna is used for fuzzy-testing over custom generated test-cases. For the given token.sol, the testToken.sol is created and the tests are written. These test functions are required to start from "echidna\_"

### D. The sum-to-n Contract





## VI. RESULT AND CONCLUSION

We conclude our study after achieving the following objectives

- Exploring on various domains of analysis like symbolic execution, static analysis
- Exploring various categories of bugs and vulnerabilities associated with smart-contracts as a whole and in depth
- Analysis of tools like Slither, Manticore, Mythril, Echidna and KEVM for the purpose of reasoning, verification and finding vulnerabilities.
- Extracting a representative set of smart-contracts for analysis

Slither, Manticore and Mythril over sample contracts were emulated over Google-Collab Notebook (Attached as *Analysis notebook*. Slither and Mythril over 143-SWC Contracts were emulated over Google Collab Notebook named as *Analysis-SWC-Contracts*. Echidna was emulated over Docker whereas KEVM was emulated over Linux Terminal

Apart from Slither, Manticore, Mythril, Echidna and KEVM, we tried to integrate and test out tools like Securify, Etheno, Rattle, and SmartCheck.

We arrive at the following conclusions for various tools over DASP vulnerabilities

like to thank for giving us the platform to acknowledge ourselves with the tools for analysis of smart-contracts

## REFERENCES

- [1] Smart Contract Vulnerability Detection Technique: A Survey. (<https://arxiv.org/pdf/2209.05872.pdf>)
- [2] Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts (<https://arxiv.org/pdf/1910.10601.pdf>)
- [3] Huang, Yongfeng, Bian, Yiyang, Li, Renpu, Zhao, J. Shi, Peizhong. (2019). Smart Contract Security: A Software Lifecycle Perspective.
- [4] Petar Tsankov, A. Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, Martin T. Vechev. Computer Science CCS (2018) Securify: Practical Security Analysis of Smart Contracts <https://arxiv.org/abs/1806.01143>
- [5] <https://swcregistry.io/>

	Re-entrancy	Integer Overflow	Access Control	Exception Handling	Denial of Service
Slither	✓				
Mythril	✓	✓		✓	✓
Echidna	✓			✓	
KEVM	Provide verification conditions for contract program analysis and formal verification methods.				
	Block Parameter Dependency	Short Address	Ether Frozen	Replay Attack	Unknown Function Call
Slither					✓
Mythril	✓		✓		
Echidna	✓		✓		
KEVM	Provide verification conditions for contract program analysis and formal verification methods.				
	Ether Loss	Call-Stack overflow tx.origin	Timestamp Dependency	Transaction Order Dependency	
Slither	✓	✓	✓	✓	
Mythril	✓		✓	✓	
Echidna				✓	
KEVM	Provide verification conditions for contract program analysis and formal verification methods.				

Fig. 18. Analysis of tools over DASP and SWC

A conclusion to draw is that the current state-of-the-art tools are still far away from achieving complete protection and hence there is a lot of scope in the same.

## ACKNOWLEDGMENT AND CONTRIBUTIONS

We would like to thank Dr Somanath Tripathy for giving us an opportunity to work on this project. The individual contributions are cited as: Kavya Goyal(1901CS30) Working of Slither, Manticore and Mythril along with working and analysis of 143-SWC contracts. Priyanka Sachan (1901CS43) Working of Echidna and Formal Verification. We would again