Verifying Security Vulnerabilities for Blockchain-based Smart Contracts

Nedas Matulevicius and Lucas C. Cordeiro

University of Manchester, UK

November 2021

Contents

- Introduction
- Aims and Objectives
- Motivation
- Contributions
- Background: testing approach, static analysis tools, tools and equipment, performance metrics and smart contract tests
- Evaluation and Analysis: description of benchmarks, setup, summary of tests, objectives, results and threats to validity
- Conclusions

Introduction

Blockchain technology becomes more popular and more people are interested in the field

New technology – new opportunities:

- Cryptocurrencies – Bitcoin, Ethereum, Litecoin etc.
- Secure sensitive data transfer
- IoT device management systems
- Online auctions
- Electronic voting systems

Blockchain technology being relatively new poses a cybersecurity challenge

Introduction

Interaction with the blockchain – through smart contracts

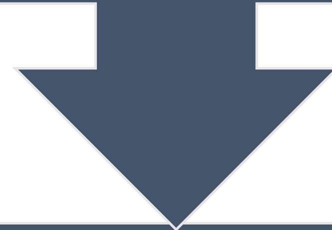
- A piece of code which handles the logic required for the intended use
- Written in languages compatible with blockchain platforms (Solidity)
- Publicly visible to anyone
- Once deployed, it is hard to remove them from the blockchain

Smart contract code can be unsafe

- Logical errors
- Uncaught exceptions
- Buffer overflow
- Unsafe usage of low-level functions

Aims and Objectives

Main objective: find out the best publicly available Solidity smart contract static analysis tool



Specific aims:

Writing various smart contracts as tests for verifiers to check their accuracy and efficiency	Finding, using and adapting tests where applicable to various existing smart contract verifiers	Performing benchmarking tests on verifiers	Performing analysis and statistics given benchmarks and verifier accuracy to derive conclusions

Motivation

Unsafe smart contract code = financial losses

- DAO attack - $50 million (2016, over $5 billion in 2021) worth of Ether stolen
- Parity wallet hacks – $50-150 million (2017, $777 million in 2021) worth of Ether stolen
- Integer overflow abuse - ~$1 million (2021) worth of Ether stolen
- 51% attack on Ethereum Classic - double-spending of tokens with value of $1.1 million (2019)

Unsafe smart contract code = system abuse and illegal exploits

- King of the Ether Throne game – possibility of taking the “throne” indefinitely
- Rubixi – a classical Ponzi scheme with a bug where users could withdraw all their fees
- GovernMental – miners can impersonate users in order to win the scheme

Contributions

Creation of Solidity smart contract tests:

- May or may not contain security vulnerabilities
- Can be verified with various static analysis tools adapted for Solidity smart contracts
- Security vulnerabilities included are evaluated against cybersecurity properties (CIA triad, SEI CERT Coding Standard)

Evaluation of state-of-the-art static analysis tools w.r.t. cybersecurity properties

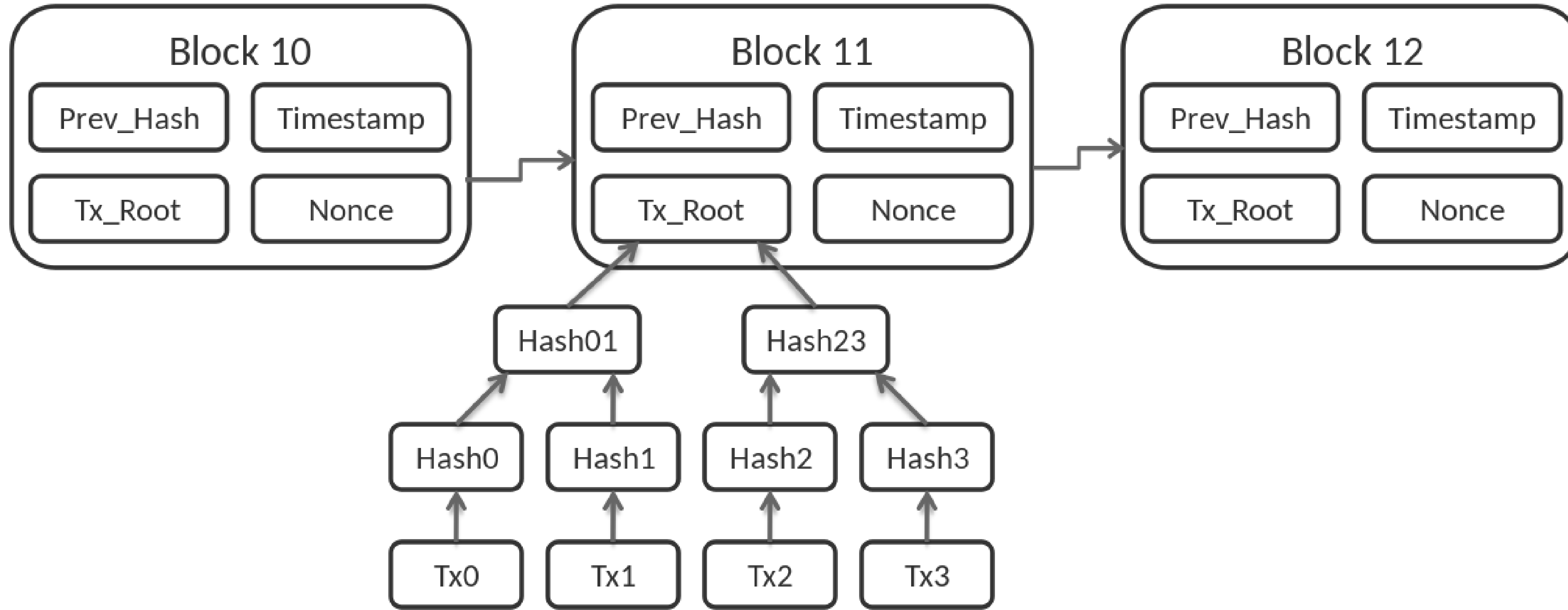
Background

Blockchain:

- Write-only list of data structures (blocks) chained together into a list (chain)
- Technology not bound to specific applications (e.g. cryptocurrency)
- Each block contains a timestamp of creation as well as hashed data
- To submit a block onto the blockchain, it needs to be approved by a majority vote (consensus mechanism)

Ethereum blockchain contains:

- States
- Transactions
- Blocks



Block 10
Prev_Hash
Timestamp
Tx_Root
Nonce
Block 11
Prev_Hash
Timestamp
Tx_Root
Nonce
Block 12
Prev_Hash
Timestamp
Tx_Root
Nonce
Hash01
Hash23
Hash0
Hash1
Hash2
Hash3
Tx0
Tx1
Tx2
Tx3

Background

Smart contracts:

- A set of rules and protocols, which are deployed in the blockchain to verify and validate the transactions between users
- Main interaction point for users with the blockchain
- Can contain malicious code and blockchain cannot protect against dangerous smart contracts
- Smart contract code is visible to anyone using the blockchain
- Ethereum smart contracts are written in Solidity
- Code can be verified with static analysis tools

Fdaf

- Fsdfds
- dsf

Smart contract static analysis tools

Smart contract verification – via static analysis tools and methods:

- E.g.: lexical and dataflow analysis, symbolic execution (symex) and model checking

Tools used:

- Remix IDE:
  - Written in JS, the Remix Analyzer for static analysis plugin is used
  - Detect-and-report approach, no complex Maths required, uses Abstract Syntax Trees (AST) for detection and evaluation of vulnerabilities
  - Plugin is fast and the IDE is easy to use, with vulnerabilities classified into several categories
- Slither:
  - Written in Python
  - Works similar to other SMT-based BMC tools (Source code -> AST -> CFG -> IR -> Code analysis and vulnerability detection -> Results printed out)
  - 75 different vulnerability detectors

Smart contract static analysis tools

Oyente:

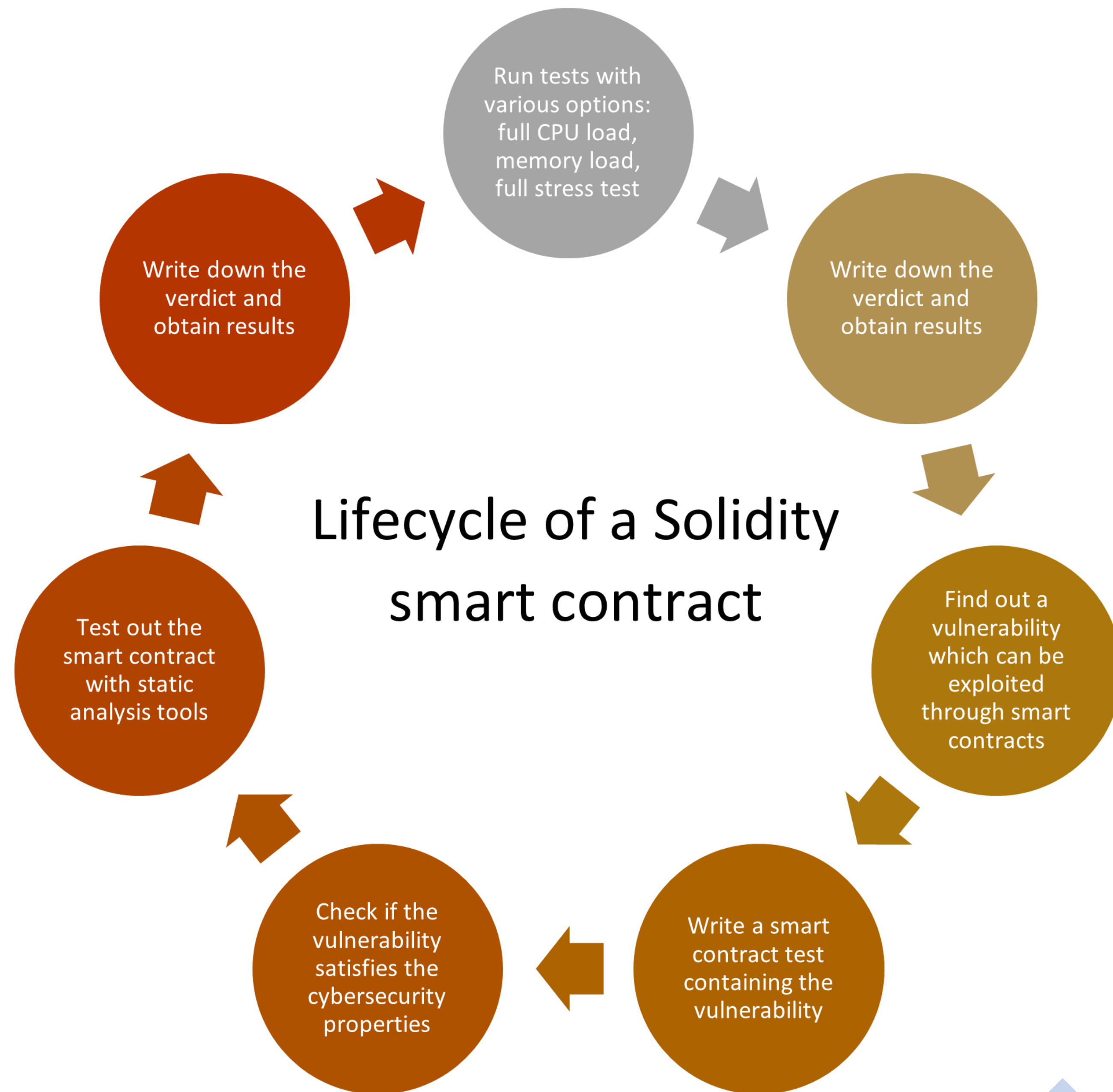
- Written in Python
- Utilises Z3 SMT solver for vulnerability detection
- Catches only 4 types of vulnerabilities: callstack, money concurrency, time dependency and reentrancy bugs
- Outdated but referenced quite often in other articles and papers

Mythril:

- Uses Z3 SMT solver as well for its custom backend, LASER-Ethereum
- Works on the same concepts as Slither and Oyente
- Mythril's vulnerability detection list is closely related to the SWC registry, which is well-documented

SmartCheck:

- Written in Java
- Uses ANTLR and a custom Solidity grammar to build its own AST, which generates an XML parse tree acting as IR
- Detect-and-report approach utilised, similar to Remix IDE plugin


Run tests with various options: full CPU load, memory load, full stress test
Write down the verdict and obtain results
Find out a vulnerability which can be exploited through smart contracts
Write a smart contract test containing the vulnerability
Check if the vulnerability satisfies the cybersecurity properties
Test out the smart contract with static analysis tools
Write down the verdict and obtain results
Lifecycle of a Solidity smart contract

Tools and equipment

Laptop for running the tests:

- Intel i7-3667U 4 Core CPU @ 2 GHz
- 8 GB of available RAM
- Hard drive of 180 GB
- Linux OS, Ubuntu distribution v. 20.04

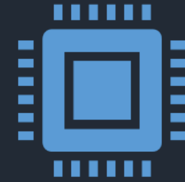
Benchmarking tools:

- htop, an improved version of in-built top tool
- hyperfine, benchmarking tool for running several tests at once with visual reports

Performance metrics

Accuracy

Speed



CPU and memory consumption

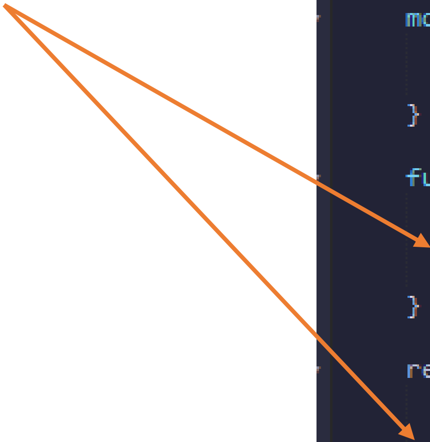
Smart contract tests

Example 1: double-spend vulnerability

- A smart contract allowing users to book a hotel room and pay for a booking in Ether
- The function *book()* checks if the user has enough money AND if the room is not occupied already
- The function *receive()* does not check for the availability -> rooms can be overbooked
- Calling *book()* and *receive()* gives only one room for the price of two!

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract Booking {

    enum State {Vacant, Occupied}
    State public state;

    address payable public owner;
    mapping (address => uint) private balance;

    event Booked(address _occupant, uint _amount);

    constructor() public {
        owner = msg.sender;
        state = State.Vacant;
    }

    modifier onlyIfVacant {
        require(state == State.Vacant, "The room is occupied!");
        _;
    }

    modifier costs(uint _amount) {
        require(msg.value >= _amount, "Insufficient funds!");
        _;
    }

    function book() public payable onlyIfVacant costs(3 ether) {
        owner.transfer(msg.value);
        state = State.Occupied;
        emit Booked(msg.sender, msg.value);
    }

    receive() external payable costs(3 ether) {
        owner.transfer(msg.value);
        state = State.Occupied;
        emit Booked(msg.sender, msg.value);
    }
```

These functions do the same operation



Checking for availability here…

… but not here

Smart contract tests

Example 2: dead code

- A smart contract which transfers Ether to owner from user after some calculations
- The function *doSomethingElse(*) takes the user's Ether into account but the result returned from that function does not go any further
- The functions *doSomethingElse()* and *doUselessCalculations()* do not contribute to the final answer and can be safely deleted
- Running the functions takes a long time and consumes a significant amount of CPU and memory

The value of variable ***answer*** is not used anywhere, the value of variable ***amount*** is

```
address payable public owner;
uint amount = 0; uint answer = 0;

event Success(address _from, uint _amount);

constructor() public {
    owner = msg.sender;
}

function doSomething() public payable {

    amount = msg.value;
    uint val = 2;

    answer = doSomethingElse(amount, val);

    owner.transfer(amount);
    emit Success(msg.sender, amount);
}

function doSomethingElse(uint a, uint b) private returns (uint value) {
    uint c = 0; uint res = 0;
    for(uint i = 0; i < b; i++) {
        if(i == 0) {
            c++;
            res += doUselessCalculations(a, b, c);
        }
        else if(a < b) {
            c += a;
            res += doUselessCalculations(b, a, c);
        }
        else {
           c++;
           res += doUselessCalculations(c, b, a);
        }
    }
    return res;
}

function doUselessCalculations(uint a, uint b, uint c) private returns (uint res) {
    uint someAnswer = 1;
    for(uint i = 1; i <= 1000; i++) {
        someAnswer += mulmod(a, b, addmod(c, c, a));
    }
    for(uint j = 1; j <= 1000; j++) {
        someAnswer += mulmod(a, b, addmod(c, 5, a));
    }
    for(uint k = 1; k <= 1000; k++) {
        someAnswer += mulmod(a, b, addmod(c, 10, a));
    }
    return someAnswer;
}
}
```

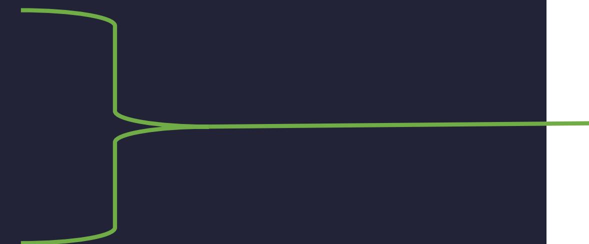
Everything else done here is just consuming the resources of the machine

Smart contract tests

Example 3: out-of-gas exceptions

- A smart contract which transfers Ether to owner from user in a loop and increments a counter
- Each operation in Solidity costs *gas,* a unit of operational costs
- Operations are limited to the amount of gas provided; if the code runs out of gas, it throws an exception
- Code not handling out-of-gas exceptions properly is prone to DDoS attacks

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract outOfGas {

    address payable public owner;
    address payable public dest;

    uint counter = 0;

     constructor() public {
        owner = msg.sender;
    }

    function setDest() public {
        dest = msg.sender;
    }


    function run() public payable {
        for(uint i = 0; i >= 0; i++) {
            dest.send(msg.value);
            counter++;
        }
    }

}
```



The loop is infinite, execution costs of this loop is also infinite

Evaluation and Analysis

All 5 static analysis tools were measured with all 13 tests

Each test was run 10 times with each static analysis tool -> 130 times for each static analysis tool -> 650 test runs in total

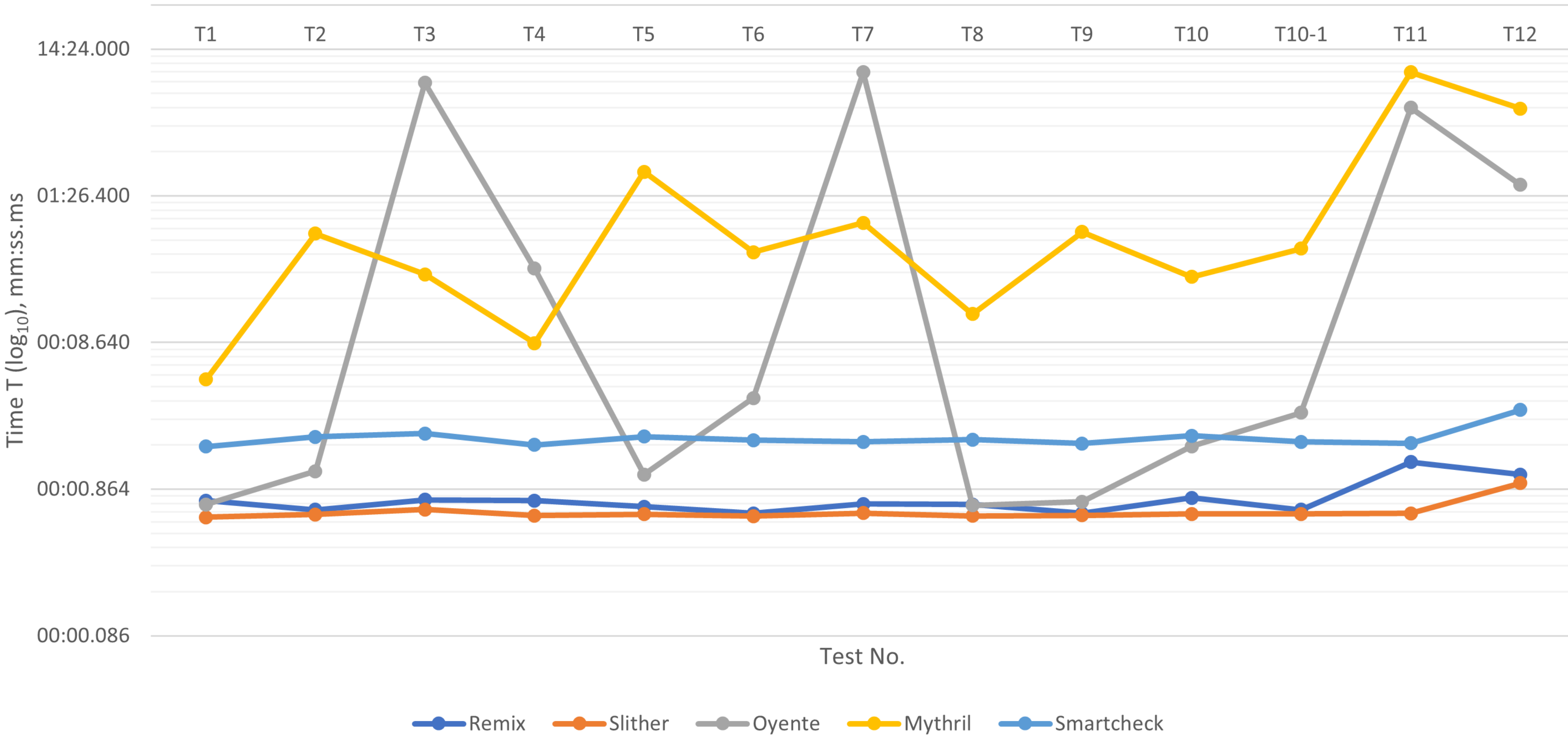
The benchmarks were divided into several parts:

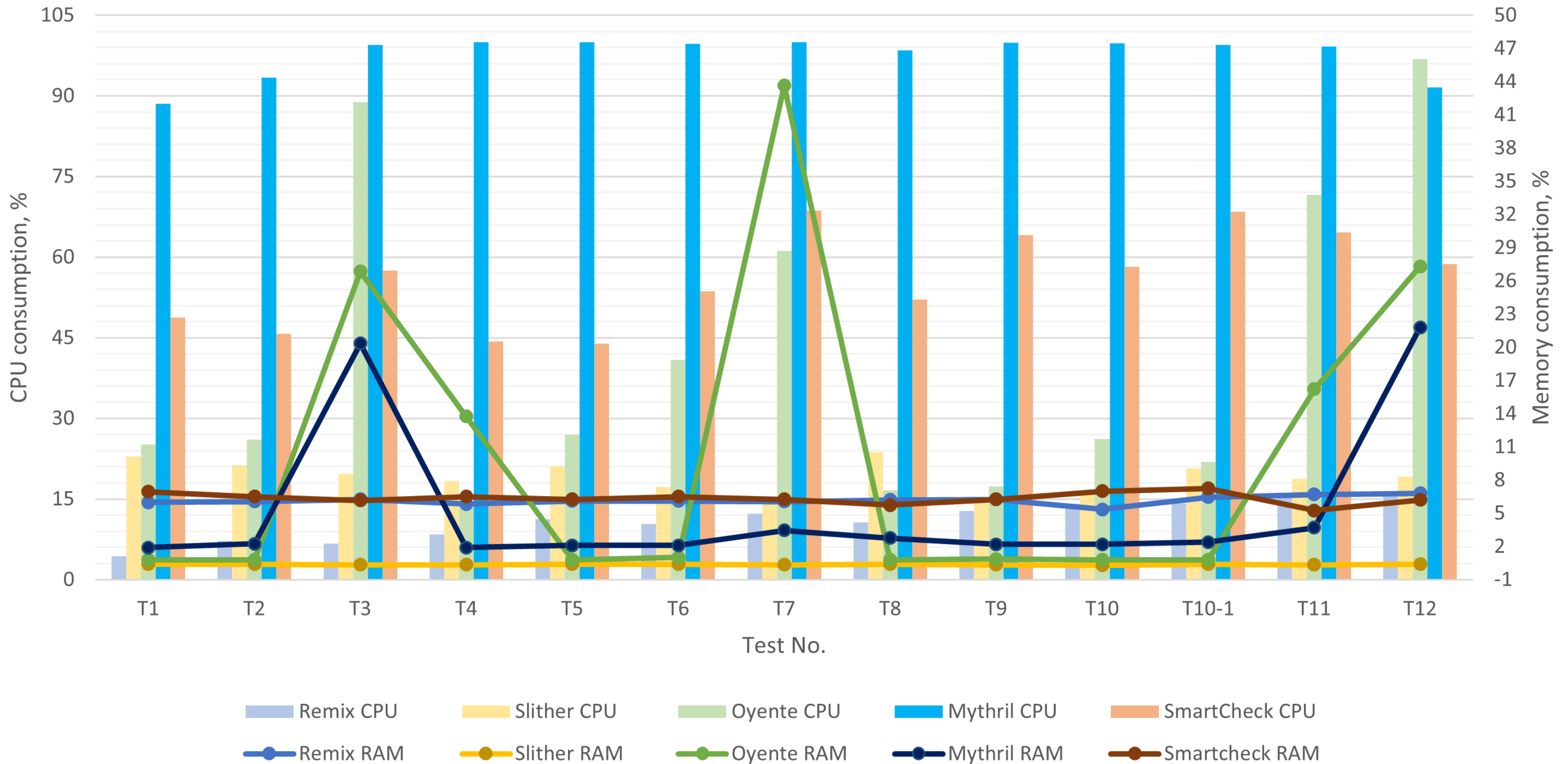
- Normal conditions – no extra load given to the computer (time and accuracy test)
- Normal conditions – no extra load given to the computer (resource management test)
- Maximum CPU load
- 77% memory load
- 90% memory load
- Maximum CPU load + 90% memory load combined

Main goal – to find out the fastest and most accurate static analysis tool

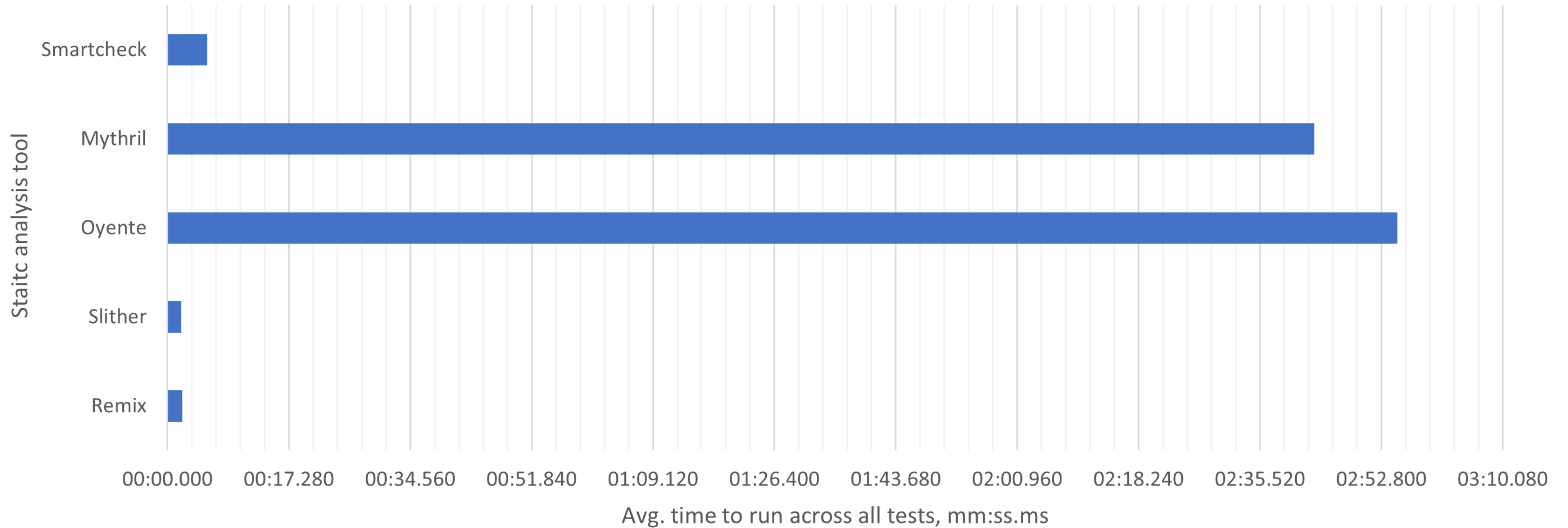
Test No.	Remix	Slither	Oyente	Mythril	SmartCheck
T1	Y	Y	Y	Y	Y
T2	Y	Y	Y	Y	N
T3	N	N	N	N	N
T4	Y	Y	N	N	N
T5	Y	Y	N	Y	N
T6	Y	Y	Y	N	N
T7	N	N	N	N	N
T8	Y	Y	N	Y	N
T9	Y	Y	N	Y	Y
T10	N	Y	N	N	N
T10.1	N	N	N	N	N
T11	Y	Y	N	N	Y
Accuracy %	66.67%	75.00%	25.00%	41.67%	25.00%

Accuracy table for each of the smart contract tests:
Y stands for “vulnerability found”, N – “vulnerability not found”


Avg. running times for each test in $\log_{10}$
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T10-1
T11
T12
14:24.000
01:26.400
00:08.640
00:00.864
00:00.086
Time T ($\log_{10}$), mm:ss.ms
Test No.
Remix
Slither
Oyente
Mythril
Smartcheck



Avg. CPU and memory consumption per test
105
90
75
60
45
30
15
0
CPU consumption, %
50
47
44
41
38
35
32
29
26
23
20
17
14
11
8
5
2
-1
Memory consumption, %
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T10-1
T11
T12
Test No.
Remix CPU
Slither CPU
Oyente CPU
Mythril CPU
SmartCheck CPU
Remix RAM
Slither RAM
Oyente RAM
Mythril RAM
Smartcheck RAM


Avg. static analysers' running time under full stress test
Staitc analysis tool
Smartcheck
Mythril
Oyente
Slither
Remix
00:00.000
00:17.280
00:34.560
00:51.840
01:09.120
01:26.400
01:43.680
02:00.960
02:18.240
02:35.520
02:52.800
03:10.080
Avg. time to run across all tests, mm:ss.ms

Results

Slither by far is the most accurate and balanced on resource management out of 5 static analysers tested

Remix IDE plugin provides good accuracy and fast results considering its approach to vulnerability detection

Mythril is resource-hungry, therefore can be prone to timeouts

SmartCheck is comparable to Mythril in accuracy, but its resource management is better

Oyente is by far the least accurate and prone to code explosion

Conclusions

Main contributions of this paper is the compilation and evaluation of Solidity smart contract tests, which fit into the cybersecurity properties

Not many surveys or analyses carried out (at least to the authors' knowledge) where smart contracts with security vulnerabilities had risk assessments or comparisons to the CIA triad – a significant achievement unique to this paper

The obtained data from the tests allow prioritisation of vulnerability checking – accuracy vs speed trade-off

Future work - extending the ESBMC to perform in-depth security analysis for Solidity smart contracts from the cybersecurity perspective