Validating Solidity Code Defects using Symbolic and Concrete Execution powered by Large Language Models

Susan Ștefan Claudiu
Arusoaie Andrei
Dorel Lucanu

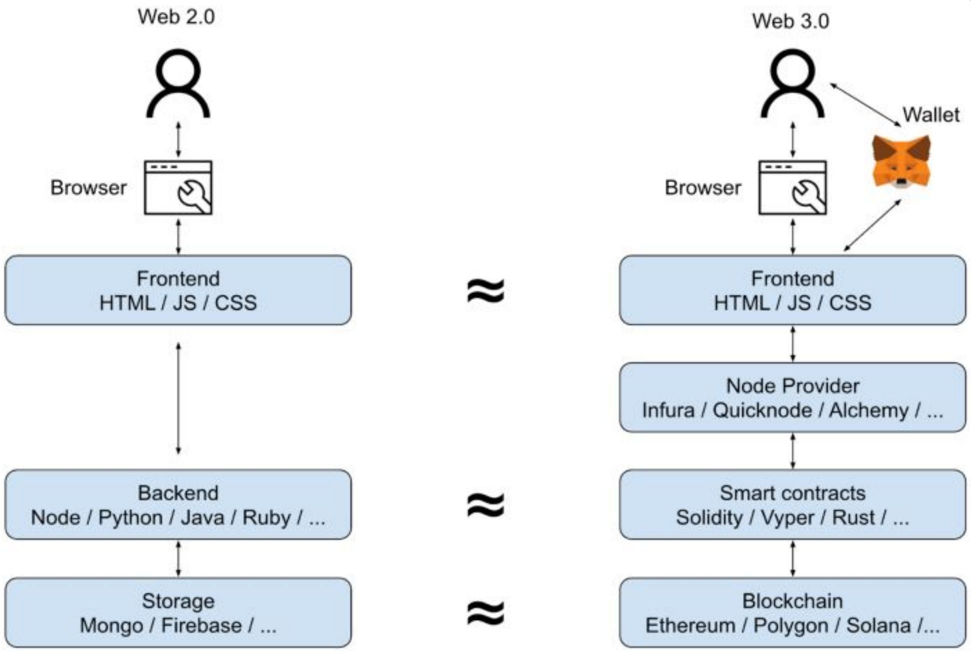
September 17 FROM 2025

Overview

- Key Terminology
- Slither and other relevant tools
- An example highlighting the limitations of Analysis Tools and LLMs
- Our detection pipeline
- An example of processing a contract using our pipeline
- Conclusion

Smart Contracts

Smart Contracts are pieces of code that run on a blockchain network. They are implemented using a programming language like Solidity, Viper, Bamboo and more.



[Yifei Huang. Decoding Ethereum smart contract data (2021)]

Solidity

According to the official documentation, Solidity is a statically typed, compiled programming language for implementing smart contracts. It was designed to target the Ethereum Virtual Machine (EVM).

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

contract Bank {
    mapping(address => uint) private balance;

    function deposit() external payable {
        balance[msg.sender] = msg.value;
    }

    function withdraw() external {
        uint addrBal = balance[msg.sender];
        payable(msg.sender).transfer(addrBal);
        balance[msg.sender] = 0;
    }
}
```

Defects in Smart Contracts are Critical

Immutable

Once a smart contract is deployed, we are unable to replace it with a newer version

Public

Even though not explicitly public, the source code of the deployed Smart Contract can still be retrieved

Financial

Most Smart Contracts directly handle a form of currency or other classes of assets

Slither

Static Analyzer for Smart Contracts

```
(env) PS E:\Contracts> slither .\GameContract.sol
'solc --version' running
'solc .\GameContract.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc
INFO:Detectors:
GameContract.play() (GameContract.sol#31-41) sends eth to arbitrary user
        Dangerous calls:
        - address(msg.sender).transfer(10 * fee) (GameContract.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
GameContract.play() (GameContract.sol#31-41) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp >= _gameStartTime && block.timestamp <= _gameEndTime (GameContract.sol#3
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
GameContract._owner (GameContract.sol#5) should be immutable
GameContract.fee (GameContract.sol#8) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Slither:.\GameContract.sol analyzed (1 contracts with 100 detectors), 4 result(s) found
```

Running Slither from the CLI

Defects found by Slither

```
import os
from slither.slither import Slither
```

```
slither = Slither(contract_path)
```

Importing the Slither module and initializing the object which gives us access to contract data

Other Development & Testing Tools for Smart Contracts

Forge

- Part of the Foundry suite along with Anvil and Cast.
- Facilitates the development and testing of Smart Contracts using only Solidity
- Allows developers perform fuzz testing on smart contracts

Kontrol

- A formal verification tool integrated within the Foundry ecosystem
- Fully compatible with existing Forge tests.
- It analyses all possible execution paths through symbolic execution

Example - Reentrancy False Alarm

A **Reentrancy** attack is a smart contract vulnerability that occurs when an external contract call is able to call back into ("re-enter") the original function before it has finished executing. However, the *transfer* function does not allow such exploits due to the limited gas that is forwards.

```
// SPDX-License-Identifier
pragma solidity 0.8.29;

contract Bank {
    mapping(address => uint) private balance;

    function deposit() external payable {
        balance[msg.sender] = msg.value;
    }

    function withdraw() external {
        uint addrBal = balance[msg.sender];
        payable(msg.sender).transfer(addrBal);
        balance[msg.sender] = 0;
    }
}
```

The balance of the caller is reduced to 0 after sending the currency.

Slither can be tricked!

```
INFO:Detectors:
Reentrancy in Bank.withdraw() (Bank.sol#11-15):
        External calls:
        - address(msg.sender).transfer(ad
        State variables written after the c
        - balance[msg.sender] = 0 (Bank.sol#14)
Reference: https://github.com/crytic/slither/wiki/Detector-Document
ation#reentrancy-vulnerabilities-4
INFO:Slither:Bank.sol analyzed (1 contracts with 100 detectors), 1
result(s) found
```

Wrongfully reporting a Reentrancy vulnerability for the contract on the previous Slide

Do LLMs fare any better?

Hello, do you think that this contract is vulnerable to reentrancy?

...

Based on the contract code you've provided, the `withdraw` function is vulnerable to a **reentrancy attack**. 😟

Even with additional help?

The transfer function only attaches 2300 gas to the transaction, is it enough to call a function?

No, 2,300 gas is generally not enough to call a function that modifies state (like another `withdraw` call), which is why the `transfer` function was historically used as a method to mitigate reentrancy attacks.

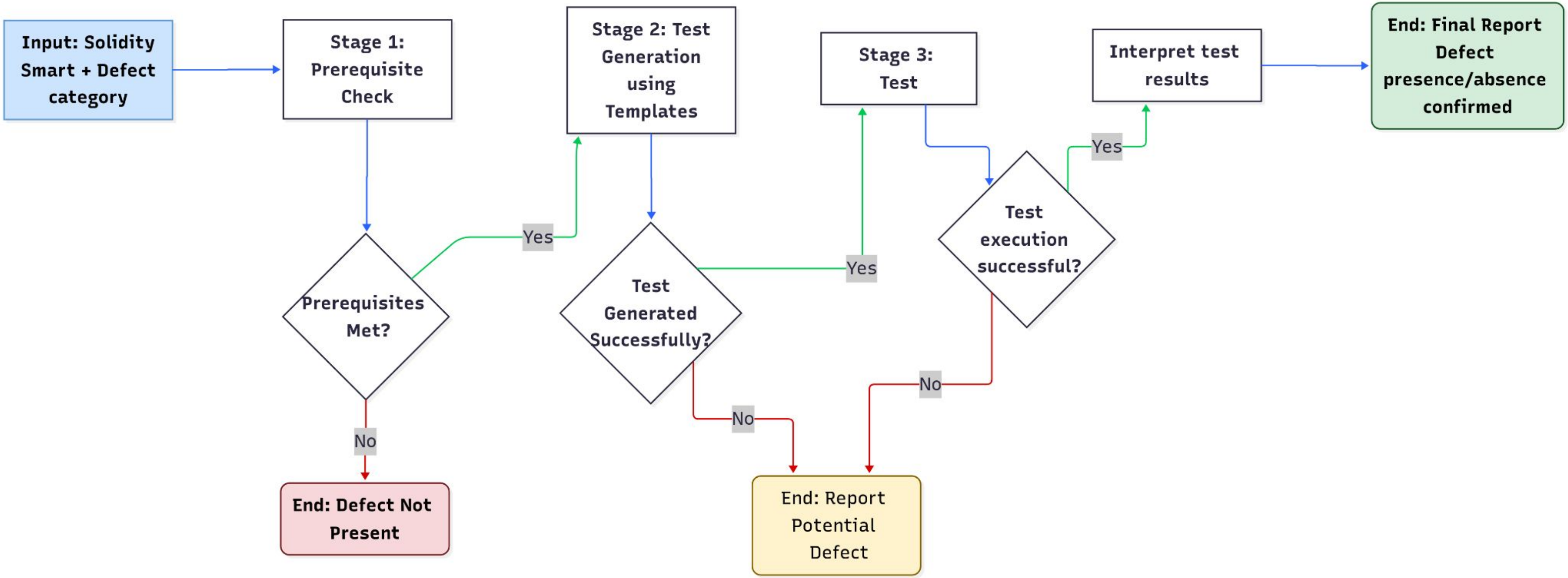
So is this contract vulnerable to Reentrancy?

Yes, the `withdraw` function in the contract you provided is vulnerable to a reentrancy attack.

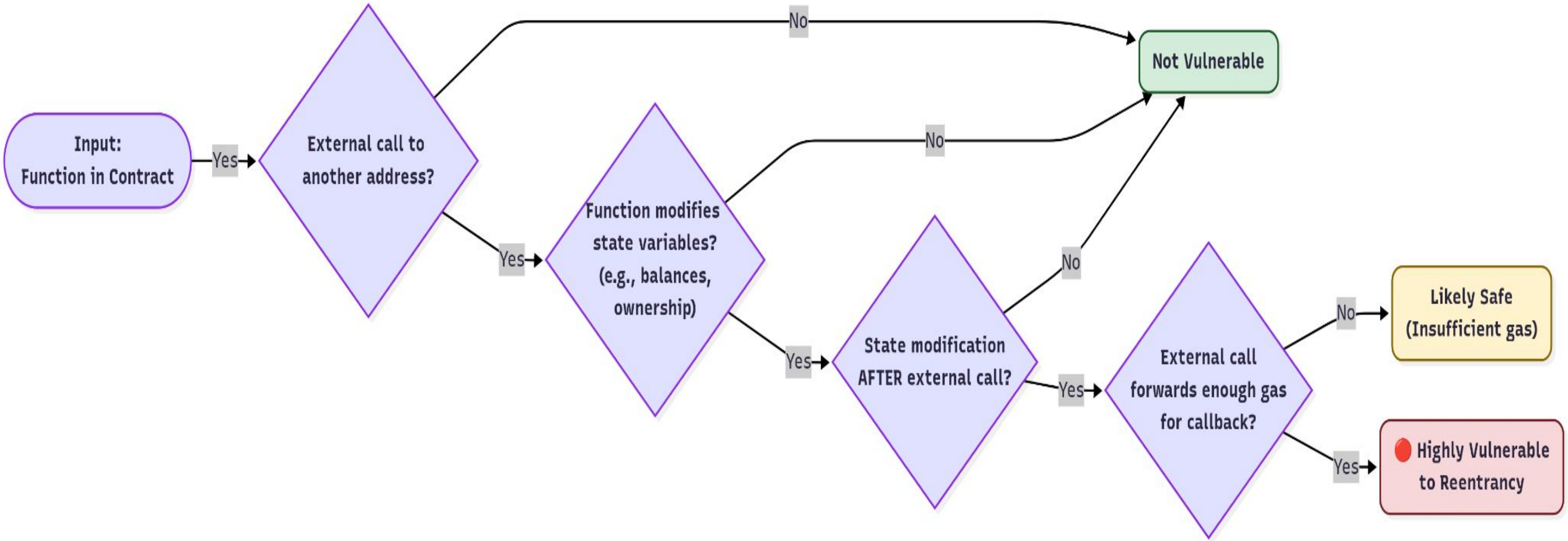
Analysis Tools vs LLMs

	Analysis Tools	LLMs
Detection Rate	Narrower detection range compared to LLMs	Able to correctly flag scenarios which Analysis Tools miss
False Alarms	No false alarms for most defect categories	Notable number of false alarms, over-generalization of training data
Resource Consumption	Most tools run on common hardware	Only small models run on common hardware
Consistency	Deterministic	Debatable

Contribution: Empower Static Analysis Tools with LLMs Inference



Example: Reentrancy prerequisites



Example: Reentrancy Test Template

```
//Append the type of the contract being tested to the name of the Test contract
contract Test is Test {
 //Replace the type "ContractUnderTest" of _contractUnderTest with the type of the contract that is currently being analyzed"
        ContractUnderTest public _contractUnderTest;
        Attacker public _attacker;

    // If the constructor of the contract under test has parameters, the "setUp" function should have the same parameters
     function setUp() public {
        //Initialize the contract being tested with the correct constructor, use the correct parameters

        //If the constructor is payable, use "vm.deal" to add balance to the current contract before intializing the contract under test

        _contractUnderTest = new ContractUnderTest();

        _attacker=new Attacker(address(_contractUnderTest));
    }

    //IMPORTANT: Do not use hardcoded parameters. Add the paramters to the "test_proofReentrancyExploit" method and forward them o the method being tested. I want to perform symbolic execution or fuzz testing using t
    //We use this function to perform that Reentrancy attacks fail against the contract
      function test_proofReentrancyExploit(uint initialDeposit) public {
        //Make sure that both contracts have enough Ether to simulate this scenario, adjust the following "vm.deal" lines if needed
        vm.deal(address(_attacker), initialDeposit);
        vm.deal(address(_contractUnderTest), initialDeposit);
        // --- ARRANGE ---
        //Use "vm.prank" to make sure that all subsequent calls will be made using the attacker's address
        vm.prank(address(_attacker));
        //Call the function from the attacker contract that prepares the attack. Attach any paramters that are needed. Make sure to not use hardcoded value
        //Add parameters to the "test_proofReentrancyExploit" if needed and forward them.
        _attacker.setupAttack();
        // --- ACT ---
        //Call the function from the attacker contract that attempts a Reentrancy attack. Attach any paramters that are needed.
        //Make sure to not use hardcoded values. Add parameters to the "test_proofReentrancyExploit" if needed and forward them.
        _contractUnderTest.attack();
        assert(address(_attacker).balance > initialDeposit);
    }

    function test_proofWithdrawUsuallyWorks(uint initialDeposit) public {
        //Make sure that both contracts have enough Ether to simulate this scenario, adjust the following "vm.deal" lines if needed
        vm.deal(address(this), initialDeposit);
        vm.deal(address(_contractUnderTest), initialDeposit);
        // --- ARRANGE ---
        //Perform any setup steps that are necessary to receive Ether from the contract being tested

        // --- ACT ---
        //Call the method that sends Ether to this contract

    }
```

Initialize the contract under test and additional elements

Check if the call is successful for Malicious users

Check if the call is successful for non Malicious users

Example: Reentrancy Test Template 1

```
//Replace this import with one correcponding to the contract type being tested,
// the file has the same name as the contract and is located in the same folder path as t
import {ContractUnderTest} from "../../src/ContractUnderTest.sol";

//Append the type of the contract being tested to the name of the Test contract
contract Test is Test {
  //Replace the type "ContractUnderTest" of _contractUnderTest with the type
  // of the contract that is currently being analyzed"
        ContractUnderTest public _contractUnderTest;
        Attacker public _attacker;

    // If the constructor of the contract under test has parameters,
    //the "setUp" function should have the same parameters
    function setUp() public {
        //Initialize the contract being tested with the correct constructor,
        // use the correct parameters

        //If the constructor is payable, use "vm.deal" to add balance to the current
        //contract before intializing the contract under test

        _contractUnderTest = new ContractUnderTest();

        _attacker=new Attacker(address(_contractUnderTest));
    }
```

Dummy import

Dummy declaration

Dummy initialization

Example: Reentrancy Test Template 2

```
//IMPORTANT: Do not use hardcoded parameters. Add the paramters to the "test_proofReentrancyExploit"
//I want to perform symbolic execution or fuzz testing using this test
//We use this function to perform that Reentrancy attacks fail against the contract
  function test_proofReentrancyExploit(uint initialDeposit) public {
    //Make sure that both contracts have enough Ether to simulate this scenario, adjust the following
    vm.deal(address(_attacker), initialDeposit);
    vm.deal(address(_contractUnderTest), initialDeposit);
    // --- ARRANGE ---
    //Use "vm.prank" to make sure that all subsequent calls will be made using the attacker's address
    vm.prank(address(_attacker));
    //Call the function from the attacker contract that prepares the attack. Attach any paramters tha
    //Add parameters from "test_proofReentrancyExploit" if needed and forward them.
    _attacker.setupAttack();
    // --- ACT ---
    //Call the function from the attacker contract that attempts                              param
    //Make sure to not use hardcoded values. Add parameters to t                              if ne
    _attacker.attack();
    // --- ACT ---
    assert(address(_attacker).balance > initialDeposit);
}
```

Perform the attack and check the balance of the attacker

Example: Reentrancy Test Template 3

```
function test_proofWithdrawUsuallyWorks(uint initialDeposit) public {
    //Make sure that both contracts have enough Ether to simulate this scenario, adjust the following "vm.dea
    vm.deal(address(this), initialDeposit);
    vm.deal(address(_contractUnderTest), initialDeposit);
    // --- ARRANGE ---
    //Perform any setup steps that are necessary to receive Ether from the contract being tested

    // --- ACT ---
    //Call the method that sends Ether to this contract

}

//Keep this so the test contract can receive Ether from the contract being tested on the "positive" scenario
receive() external payable {
}
```

Example: Reentrancy Generated Test

```
import {Bank} from "../../src/Bank.sol";

contract BankTest is Test {
        Bank public _contractUnderTest;
        Attacker public _attacker;

    function setUp() public {
        _contractUnderTest = new Bank();
        _attacker = new Attacker(address(_contractUnderTest));
    }

    function test_proofReentrancyExploit(uint initialDeposit) public {
        vm.deal(address(_attacker), initialDeposit);
        vm.deal(address(_contractUnderTest), initialDeposit);
        // --- ARRANGE ---
        vm.prank(address(_attacker));
        _attacker.setupAttack(initialDeposit);
        // --- ACT ---
        _attacker.attack();
        assert(address(_attacker).balance > initialDeposit);
    }

    function test_proofWithdrawUsuallyWorks(uint initialDeposit) public {
        vm.deal(address(this), initialDeposit);
        vm.deal(address(_contractUnderTest), initialDeposit);
        // --- ARRANGE ---
        _contractUnderTest.deposit{value: initialDeposit}();
        // --- ACT ---
        _contractUnderTest.withdraw();
    }
```

Concrete test contract setup generated by the model

Method usage scenario generated by the model

Example: Reentrancy Attacker Template

```
//Add the preparation steps necessary to perform the attack in this method,
//add any parameters that are needed, do not use hardcoded values.
function setupAttack() public payable{
    attackCallCount=0;
}

//Call the reentrant method, add any parameters that are needed,
//do not use hardcoded values.
function attack() public{

}

//We only want to call the function once more.
//This is enough to prove Reentrancy exploits without risking running out of gas
receive() external payable {
    if (attackCallCount < 1) {
        attackCallCount++;
        //Add a call to the reentrant method here to perfo
        //If paramters are needed, set them using state variables before performing the attack.
    }
}
```

Attack setup step, model must fill additional setup steps

Empty attack scenario, must be filled by the model

receive() implementation must facilitate reentrant calls

Example: Reentrancy Attacker Generated

```
function setupAttack(uint initialDeposit) public payable {
    attackCallCount = 0;
    _victim.deposit{value: initialDeposit}();
}

function attack() public {
    _victim.withdraw();
}

receive() external payable {
    if (attackCallCount < 1) {
        attackCallCount++;
        _victim.withdraw();
    }
}
```

Attack setup step generated by the model

Method under test call generated by the model

Method under test **reentrant** call generated by the model

Challenges & Limitations Documented during our Experiments

Challenges	Limitations
The cost of using LLMs via API	Forge & Kontrol integration
Receiving a structured output from LLMs	The detection of some defect categories is limited by the scope of our test templates
Receiving consistent outputs from LLMs	

Conclusion

Key Takeaways

- Novel detection pipeline: static analysis + LLMs + symbolic/concrete execution.
- Our approach effectively validates true positives.
- Eliminates false alarms that plague existing tools.

Future Work

- Extend the set of defect templates for additional defect categories to expand detection range
- Optimize and improve the test generation process
- Experiment with advanced prompting techniques and locally deployed models.

THANK YOU!

Q&A