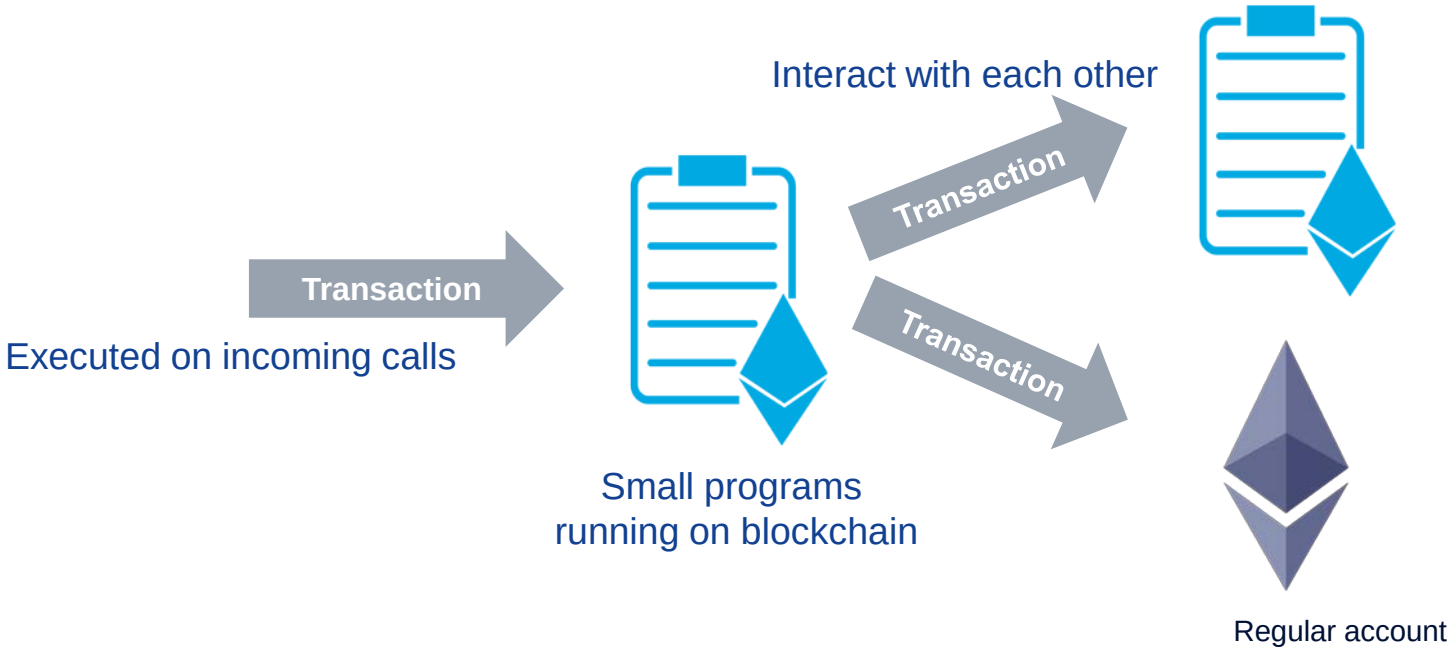
Static Analysis-Based Approaches for Enhancing Security of Smart Contracts

Asem Ghaleb

aghaleb@ece.ubc.ca

April 5th, 2022

Smart contracts



Smart contracts

High-level Turing-complete languages (e.g., Solidity)

```
Contract Bank{
    struct Account {
        address addr;
        uint balance;
    }

    Account [] accounts;

    function applyinterest () public returns (uint) {
        for (uint i =0; i< accounts.length; i++) {
            // apply 5 percent interest
            accounts[i].balance = accounts[i].balance * 105 /100;
        }
        return accounts.length
    }

    function openAccount () payable public returns (uint) {
        accounts.push(msg.sender, msg.value);
    }
}
```

Compiled

EVM Bytecode

```
PUSH1 0x80
PUSH1 0x40
MSTORE
CALLVALUE
DUP1
ISZERO
PUSH2 0x10
JUMPI
PUSH1 0x0
DUP1
REVERT
JUMPDEST
POP
PUSH2 0x289
DUP1
PUSH2 0x20
.
.
.
```

Smart contracts

- Cannot be updated
- Transactions are immutable
- Financial nature (incentive for attackers)

(2016) The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

(2017) Yes, this kid really just deleted $300 MILLION by messing around with Ethereum's smart contracts

(2019) Ethereum Classic's '51% Attack,' $1 Million Loss, Raise Concerns About Security

Smart contracts: Example

```
contract MyWallet {
  address owner;
  constructor public {
      owner = msg.sender;
  }

  function sendTo(address receiver, uint amount)

  {

    require(tx.origin == owner);

    receiver.send(amount);

  }

}
```

Wrong Authentication

Unhandled Exceptions

Other known bugs

DoS

Overflow Underflow

Locked Ether

Reentrancy

Uncheck Send

Ethereum Smart Contract Best Practices: https://consensys.github.io/smart-contract-best-practices

Overview

Code vulnerabilities are reported frequently

No evaluation methodology of analyzers

SolidiFI

False negatives

False positives

Misidentification of dugs

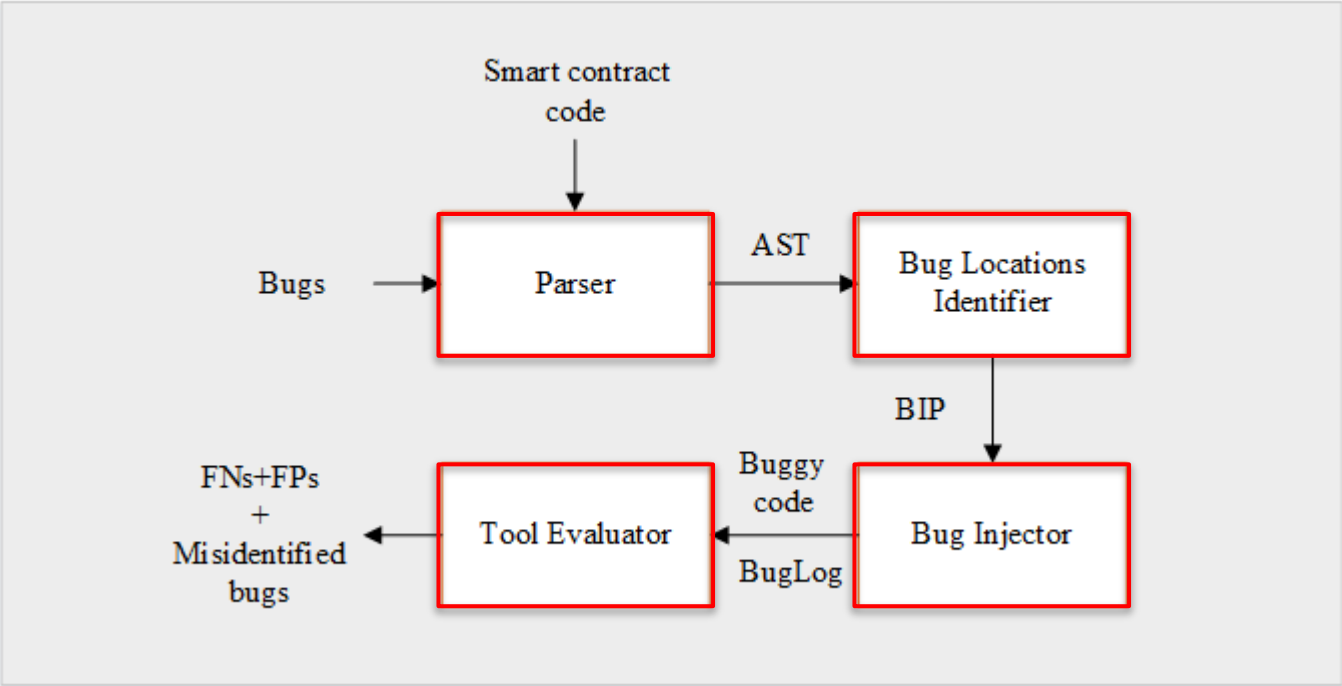
How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Analysis Tools using Bug Injection

Asem Ghaleb and Karthik Pattabiraman

Our goal

A systematic approach for evaluating efficacy of smart contract static analysis tools on detecting bugs

Approach: SolidiFI



Bug model

- Injecting code snippets which lead to vulnerabilities
- Injecting bugs claimed to be detected
- Playing the role of developers rather attackers
- Injecting distinct bugs as possible

1

```
if (startTime+5 == block.timestamp)
  { //code }
```

2

```
uint _vtime = block.timestamp;
if (startTime+5 == _vtime)
  { //code }
```

Bug injection

```
contract MyWallet {

    address owner;
    mapping(address => uint256) balances;

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

    function sendTo(address payable receiver, uint8 amount) public
    {
        require(msg.sender== owner);
        (bool success) = receiver.send(amount);
        if(!success)
          // revert();
    }

   function bug_reEntrancy ( uint256 _Amt ) public {
     require(balances [msg.sender] >= _Amt);
      (bool success,) = msg.sender.call.value(_Amt)("");
      require(success);
      balances [msg.sender] -= _Amt ;
   }
}
```

1. Code transformation
2. Security weakening
3. code snippet injection

Ethereum Smart Contract Best Practices: https://consensys.github.io/smart-contract-best-practices

Contributions

- Proposed **SolidiFI** approach
- Implemented in an automated tool
- Injected 9,369 distinct bugs
- Evaluated 6 well-known static tools

Bug Class	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	*	*	*	*	*	*
Timestamp dependency	*		*	*		*
Unchecked send		*	*			
Unhandled exceptions	*	*	*	*		*
TOD	*	*				
Integer over/underflow	*		*	*	*	
Use of tx.origin			*	*		*

Source code: https://github.com/DependableSystemsLab/SolidiFI

Findings summary

- None of the tools detected all bugs
- Many undetected corner cases
- Misidentification of bugs is high as well
- All tools reported false positives (2 to 801)
- High false positives for tools with low false negatives