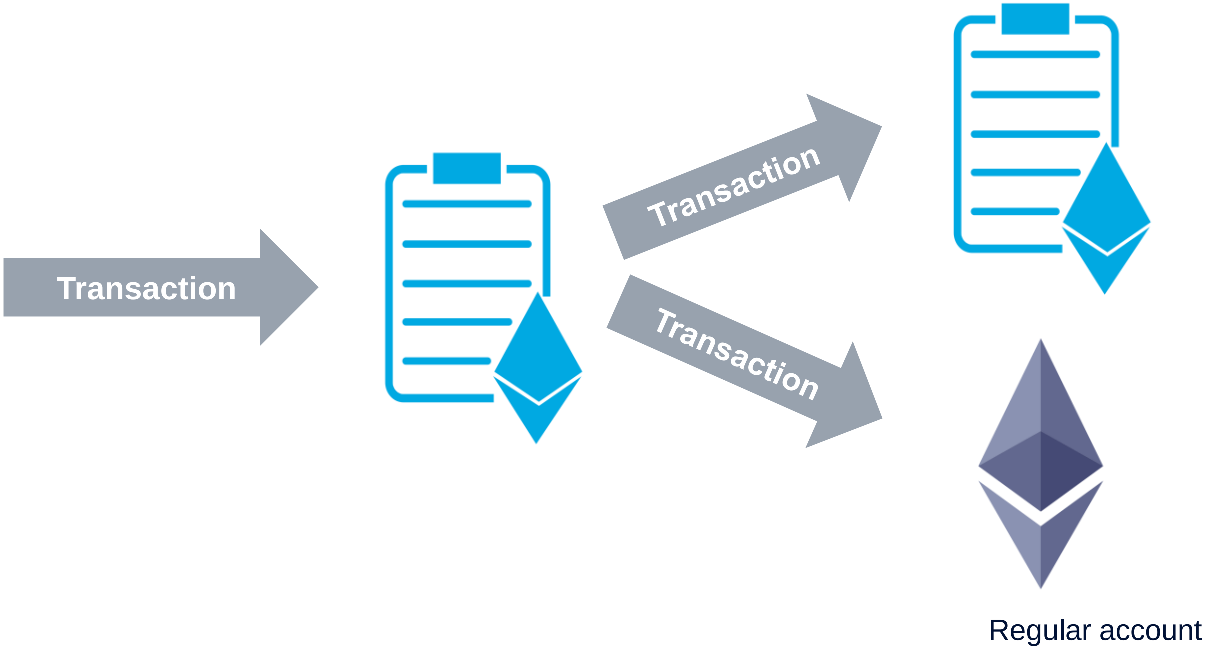
How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Analysis Tools using Bug Injection

Asem Ghaleb and Karthik Pattabiraman

July 22nd, 2020

Smart contracts



Motivation: 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

Our goal

- Code vulnerabilities are still reported frequently [1]
- No evaluation methodology of static analyzers

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

[1] S. Hwang and S. Ryu. 2020. Gap between Theory and Practice : An Empirical Study of Security Patches in Solidity. 2020. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE).

Contributions

- Systematic approach: **SolidiFI**
- Evaluated 6 static analyzers
- Analysis of the analyzers’ false negatives and false positives

Tools failed to detect several bugs and reported high false positives

Research challenges

- Solidity; different from traditional languages
- Injecting bugs into all potential locations
- Injecting exploitable vulnerabilities

Bug model

- 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

SolidiFI works on AST-level of the source code

```
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 ;
   }
}
```

Code transformation

Security weakening

code snippet injection

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

SolidiFI evaluation

- 6 static analysis tools

  (*Oyente*, *Securify*, *Mythril*, *Smartcheck*, *Manticore*, *Slither*)

- 50 Smart Contracts representative of Etherscan (39-741 loc) ~ *Most Etherscan contracts size <1000 loc*
- Different functionalities and syntactic elements

RQ1: False negatives of the evaluated tools?

RQ2: False positives of the evaluated tools?

RQ3: Injected bugs can be activated?

Experimental setup

- 7 common bug classes considered by the tools
- 9,369 distinct bugs
- Timeout: 15 minutes per smart contract

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

RQ1: False negatives of the evaluated tools

Not supported by the tool

Undetected bugs

100% detection

Security bug	Injected bugs	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	1343	1008 (844)	232 (232)	1085 (805)	1343 (106)	1250 (1108)	✓
Timestamp dependency	1381	1381 (886)	NA	810 (810)	902 (341)	NA	537 (1)
Unchecked send	1266	NA	499 (449)	389 (389)	NA	NA	NA
Unhandled exceptions	1374	1052 (918)	673 (571)	756 (756)	1325 (1170)	NA	457 (128)
TOD	1336	1199 (1199)	263 (263)	NA	NA	NA	NA
Integer over/underflow	1333	898 (898)	NA	1069 (932)	1072 (1072)	1196 (1127)	NA
Use of tx.origin	1336	NA	NA	445 (445)	1239 (1120)	NA	✓

- None of the tools detect all bugs
- Many undetected corner cases
- Misidentification is high as well

Misidentification of bugs: Example

Injected Reentrancy bug

```
142   string public name;                        //fancy name: eg Simon Bucks
143   mapping(address => uint) balances_re_ent38;
144 function withdrawFunds_re_ent38 (uint256 _weiToWithdraw) public {
145         require(balances_re_ent38[msg.sender] >= _weiToWithdraw);
146         // limit the withdrawal
147         require(msg.sender.send(_weiToWithdraw));  //bug
148         balances_re_ent38[msg.sender] -= _weiToWithdraw;
149     }
150   uint8 public decimals;                     //How many decimals to show.
```

Buggy contract

```
[
 "/tmp/SCs/buggy_1.sol:147:17: Warning: Transaction-Ordering Dependency.\n        require(msg.sender.send(_weiToWithdraw)"
]
],
```

Oyente Scan report

Reported as TOD bug

RQ2: False positives of the evaluated tools

Challenges:

- Lack of ground truth
- Large number of bugs

Approach:

Assuming a bug reported by the majority of the tools cannot be false positive

Reported Reentrancy **100** → **Filtered** **40** → **Manually inspected** **20** → **Indeed FPs** **16 80%**

Reported by majority **60**

FPs = Filtered X Indeed FPs
FPs = 40 X 80% = 32

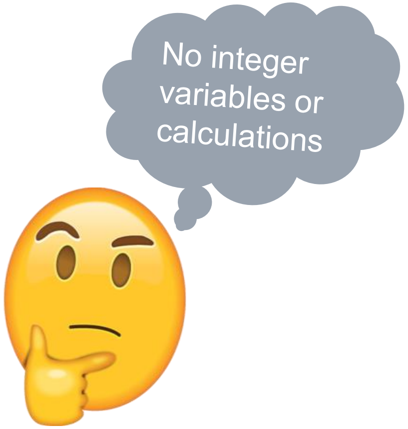
Risk: There might be false positives reported by the majority

False positive results

- All tools reported false positives (2 to 801)
- High false positives for tools with low false negatives (e.g., Slither)
- Some cases are truly bizarre

string public symbol = "CRE";

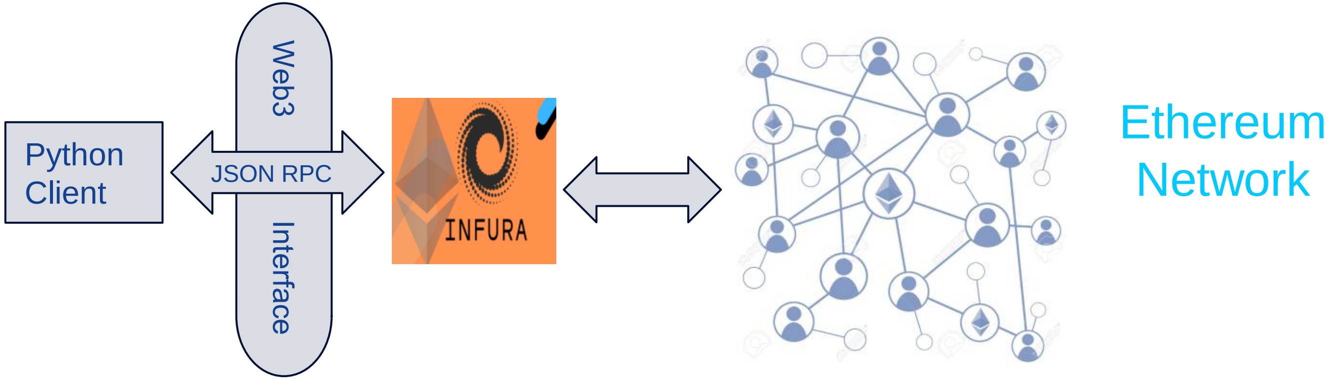
Reported as
integer overflow



RQ3: Activating the undetected bugs

Goal: Checking exploitability of the undetected bugs

- Selected 5 undetected bugs for each bug type
- All bugs were exploitable
- No much effort to exploit bugs (within minutes)



Threats to validity

- External:
  - 50 smart contracts
- Internal:
  - Evaluating 6 tools
  - 7 bug types
- Results measurement:
  - Unexploitable bugs in practice
  - True bugs counted as false positives

Summary

Goal: A systematic approach for evaluating static analyzers

- Introduced SolidiFI, for evaluating smart contract static analyzers
- Static analyzers suffer high false-negatives and false-positives
- Analyzers that **detect bugs** with **low false positives** are needed

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

Artifact: https://github.com/DependableSystemsLab/SolidiFI-benchmark

Asem Ghaleb, PhD Candidate at University of British Columbia

aghaleb@ece.ubc.ca