Blockchain Security Smart Contract Analysis

Yue Duan

Outline

- Research paper:
  - Making Smart Contracts Smarter
  - Securify: Practical Security Analysis of Smart Contracts

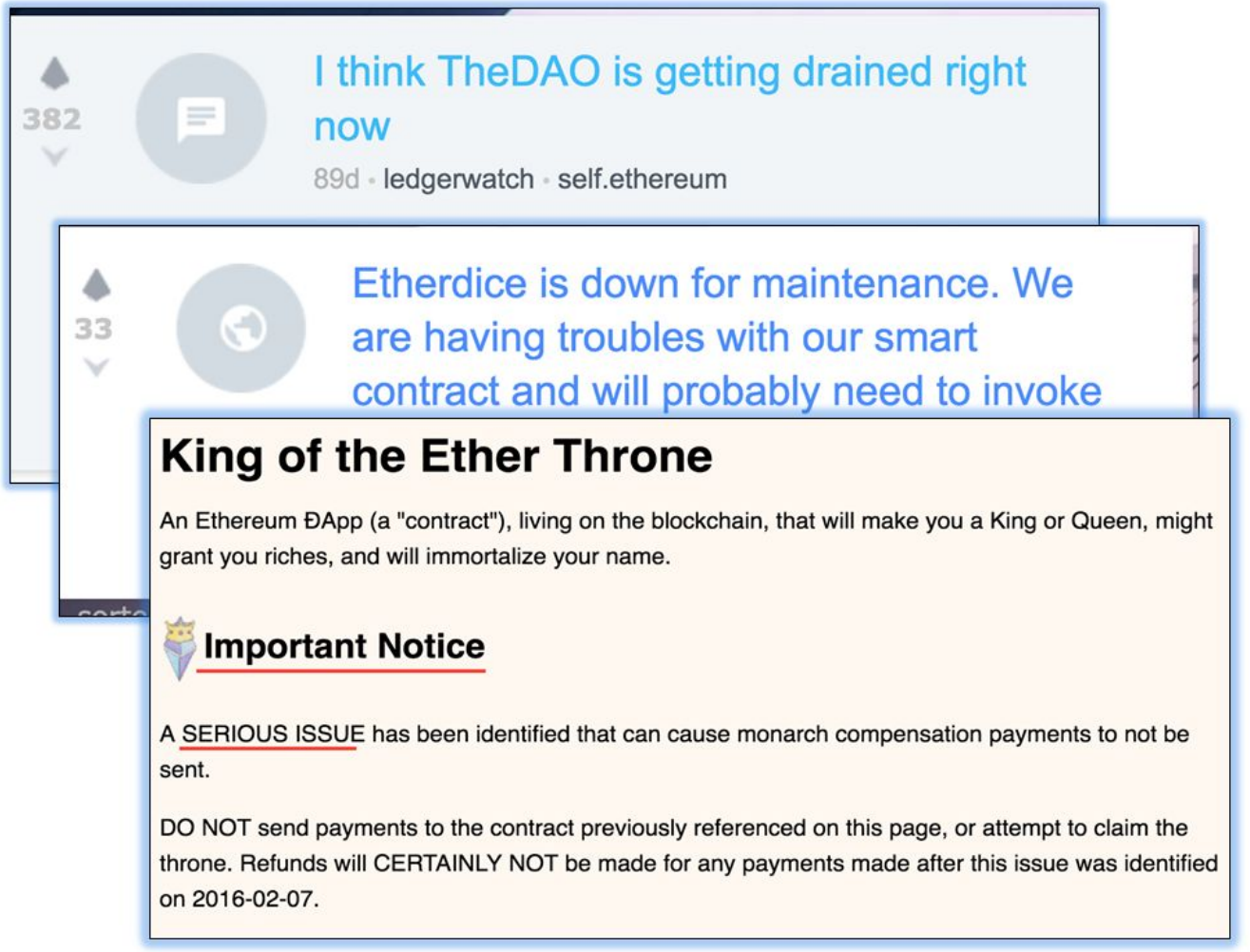
Making Smart Contracts Smarter

Loi Luu, Duc-Hiep Chu, Hrishi Olickel Prateek Saxena, Aquinas Hobor

CCS 2016

Programming Secure Smart Contracts is Hard

- Smart Contract != normal programs
  - self-executed
  - one-shot programs
- New language
  - solidity != Javascript
  - serpent != python


382
I think TheDAO is getting drained right now
89d • ledgerwatch • self.ethereum
33
Etherdice is down for maintenance. We are having troubles with our smart contract and will probably need to invoke
King of the Ether Throne
An Ethereum ĐApp (a "contract"), living on the blockchain, that will make you a King or Queen, might grant you riches, and will immortalize your name.
Important Notice
A SERIOUS ISSUE has been identified that can cause monarch compensation payments to not be sent.
DO NOT send payments to the contract previously referenced on this page, or attempt to claim the throne. Refunds will CERTAINLY NOT be made for any payments made after this issue was identified on 2016-02-07.

Introduction

- Apart from call-stack and reentrancy, are there other bugs?
- How many contracts are vulnerable?

Challenges

- Source-code may not be always available

```
1  contract Greetings {
2      string greeting;
3      function Greetings (string _greeting) public {
4          greeting = _greeting;
5      }
6
7      /* main function */
8      function greet() constant returns (string) {
9          return greeting;
10     }
11 }
```

```
6060604052604051610250380380610250833981016040528.....
```

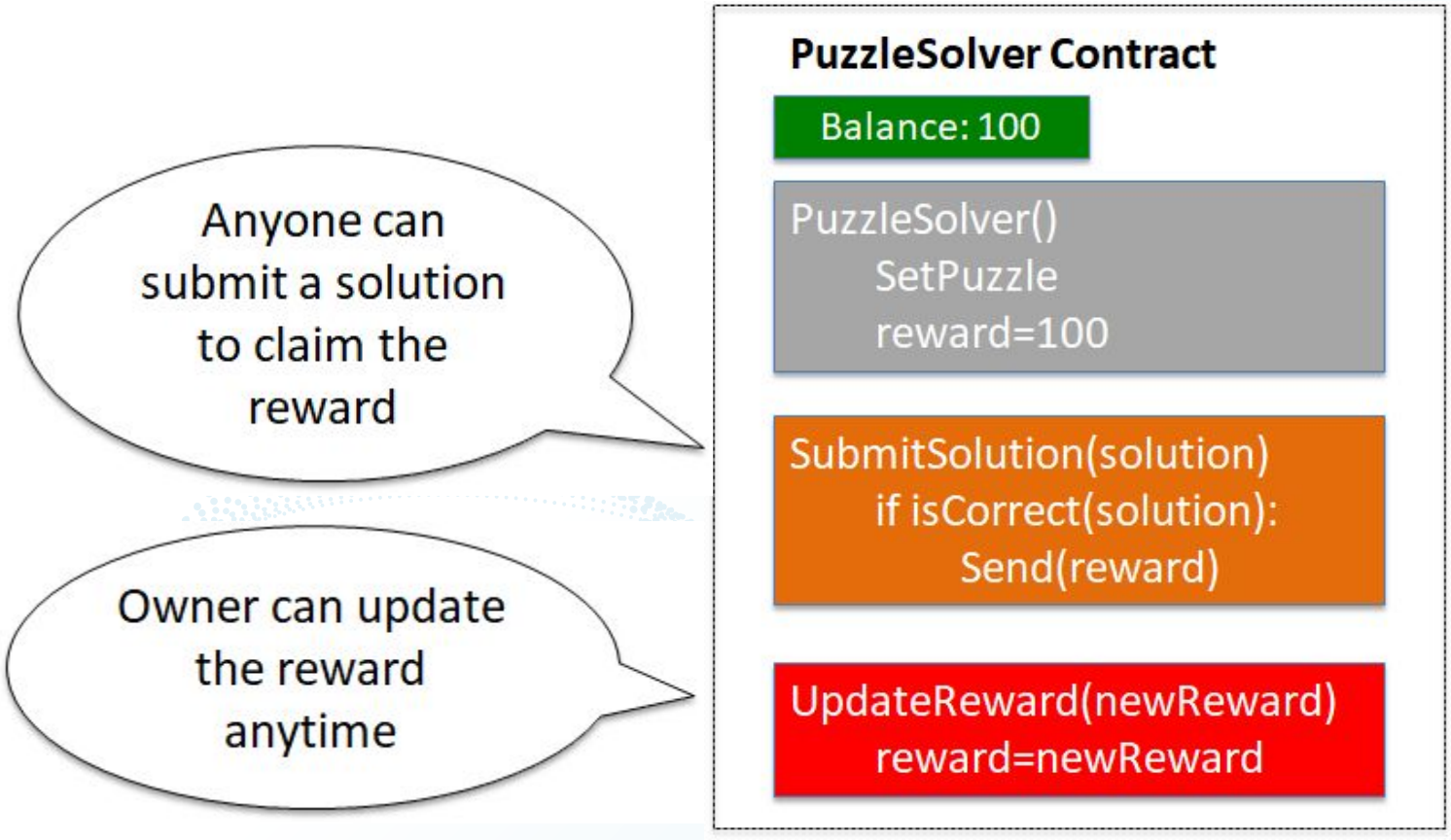
```
PUSH 60
    PUSH 40
    MSTORE
    PUSH 0
    CALLDATALOAD
    PUSH 100000000000...
    SWAP1
    DIV
.....
```

- Too many contracts
  - manual analysis is impractical

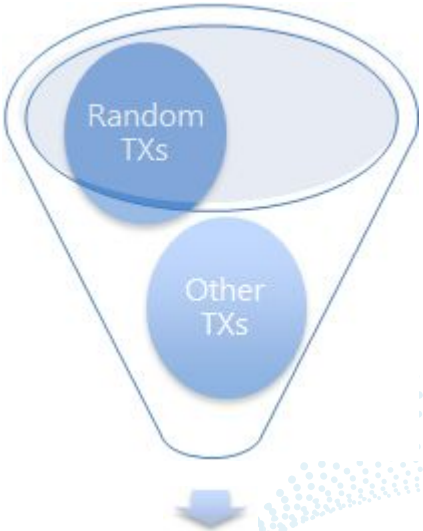
Contributions

- Identify new bugs
  - transaction ordering dependence
  - timestamp dependence
- Oyente: smart contract analyzer
  - symbolic execution tool
  - detect all popular bugs
    - TOD
    - timestamp dependence
    - reentrancy
    - mishandling exceptions (e.g. send)

Transaction Ordering Dependence



Scenario 1



Miners

Block

Random TXs

Other TXs

PuzzleSolver Contract

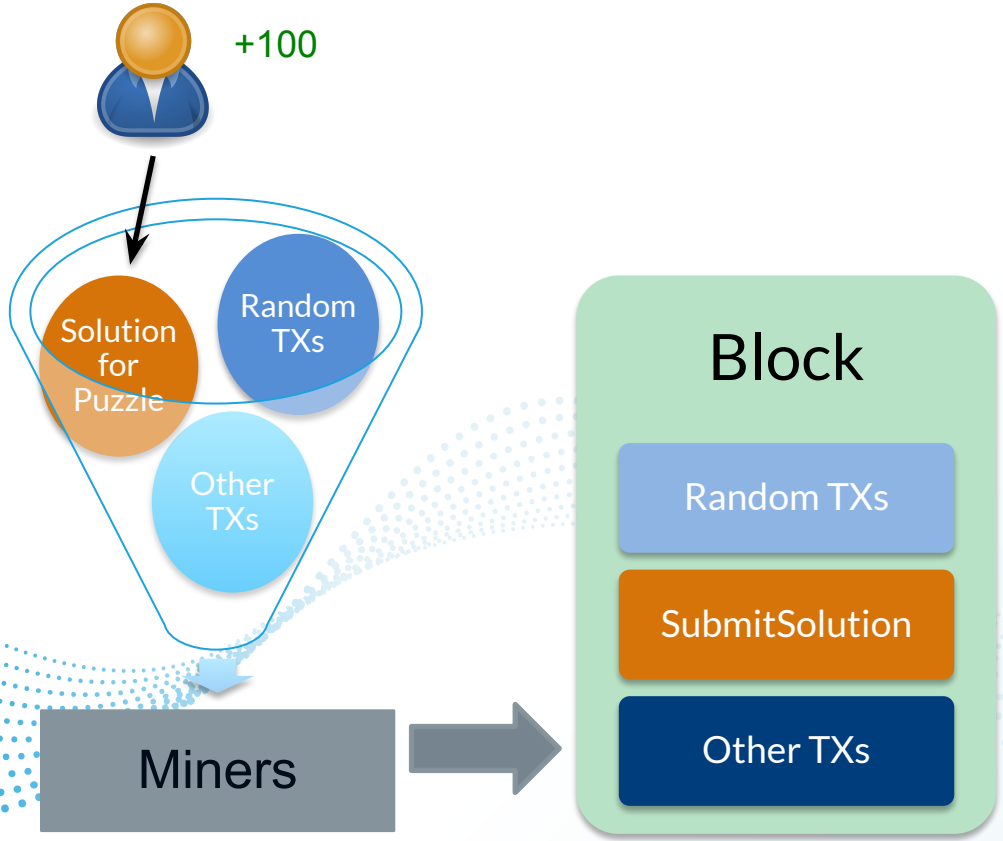
Balance: 100

```
PuzzleSolver()
    SetPuzzle
    reward=100
```

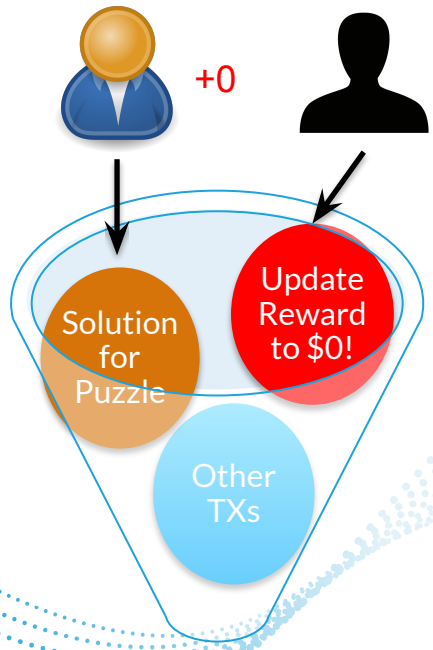
```
SubmitSolution(solution)
    if isCorrect(solution):
        Send(reward)
```

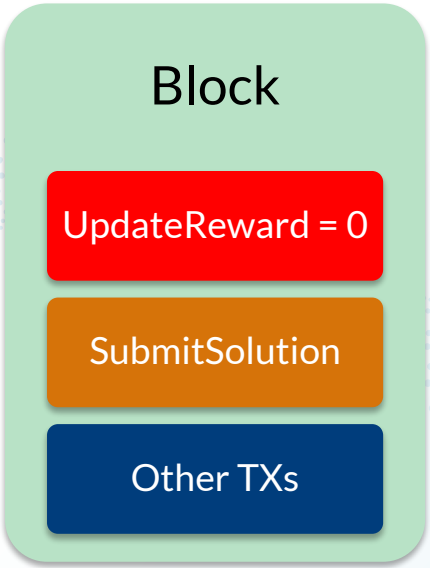
```
UpdateReward(newReward)
    reward=newReward
```

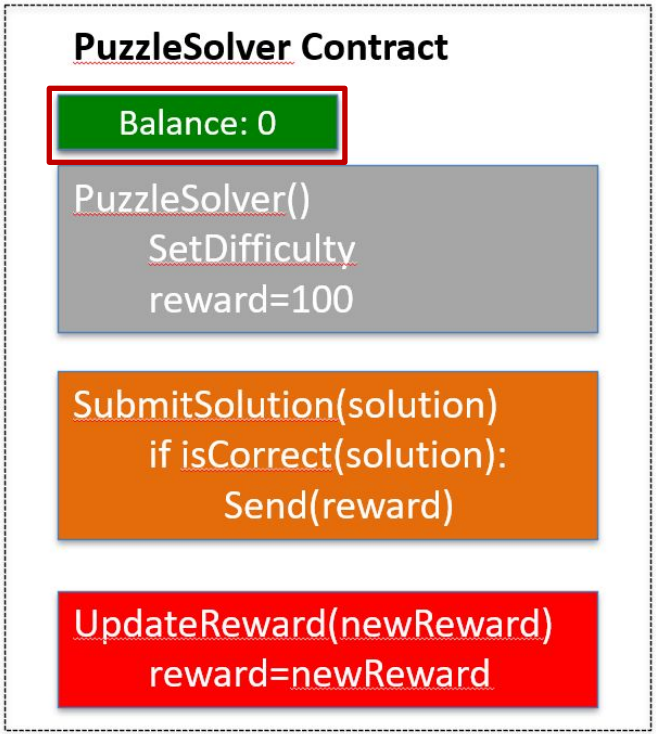
Scenario 2


+100
Solution for Puzzle
Random TXs
Other TXs
Miners
Block
Random TXs
SubmitSolution
Other TXs
PuzzleSolver Contract
Balance: 0
PuzzleSolver()
SetDifficulty
reward=100
SubmitSolution(solution)
if isCorrect(solution):
Send(reward)
UpdateReward(newReward)
reward=newReward

Scenario 3

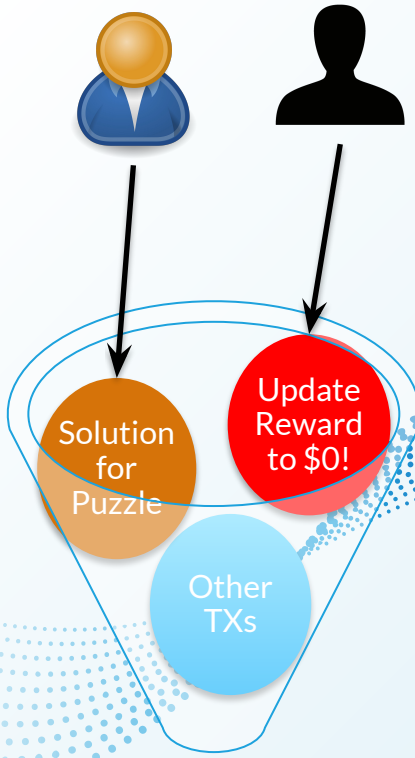

+0
Solution for Puzzle
Update Reward to $0!
Other TXs
Miners


Block
UpdateReward = 0
SubmitSolution
Other TXs


PuzzleSolver Contract
Balance: 0
PuzzleSolver()
SetDifficulty
reward=100
SubmitSolution(solution)
if isCorrect(solution):
Send(reward)
UpdateReward(newReward)
reward=newReward

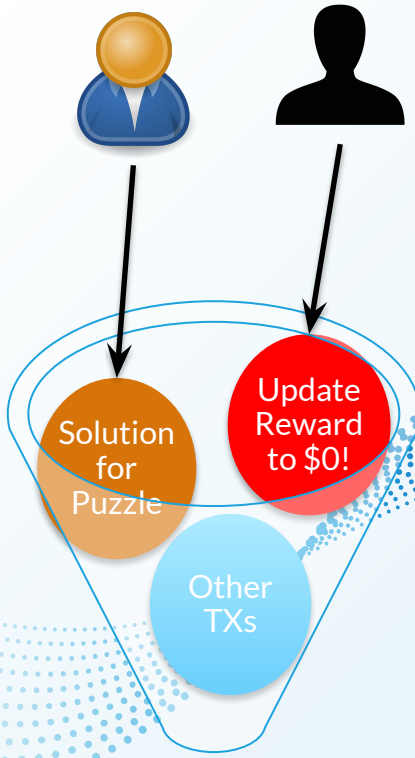
Transaction Ordering Dependence

- Observed state != Execution state
  - The expectation of the state of the contract may not be true during execution
  - Miners decide the order of TXs
- Can be coincidence
  - Two transactions happen at the same time

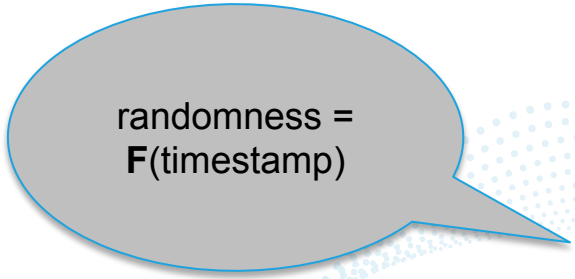


Transaction Ordering Dependence

- Can be malicious
  - Saw the targeted TX from the victim
  - Submit the second TX to update the reward
  - Both TXs enter the race



Timestamp Dependence




```
contract theRun {
  uint private LastPayout = 0;
  uint256 salt =  block.timestamp;
  function random() returns (uint256 result){
    uint256 y = salt * block.number/(salt%5);
    uint256 seed = block.number/3 + (salt%300)
                 + LastPayout +y;

    //h = the blockhash of the seed-th last block
    uint256 h = uint256(block.blockhash(seed));

    //random number between 1 and 100
    return uint256(h % 100) + 1;
  }
  ...
}
```

Timestamp Dependence

```
1  function lendGovernmentMoney(address buddy)
2      returns (bool) {
3      ...
4      if (lastTimeOfNewCredit + TWELVE_HOURS >
5          block.timestamp) {
6          msg.sender.send(amount);
7          // Sends jacpot to the last creditor
8          creditorAddresses[nCreditors - 1]
9              .send(profitFromCrash);
10         owner.send(this.balance);
11     ...
12     }
13 }
```

Timestamp is not reliable

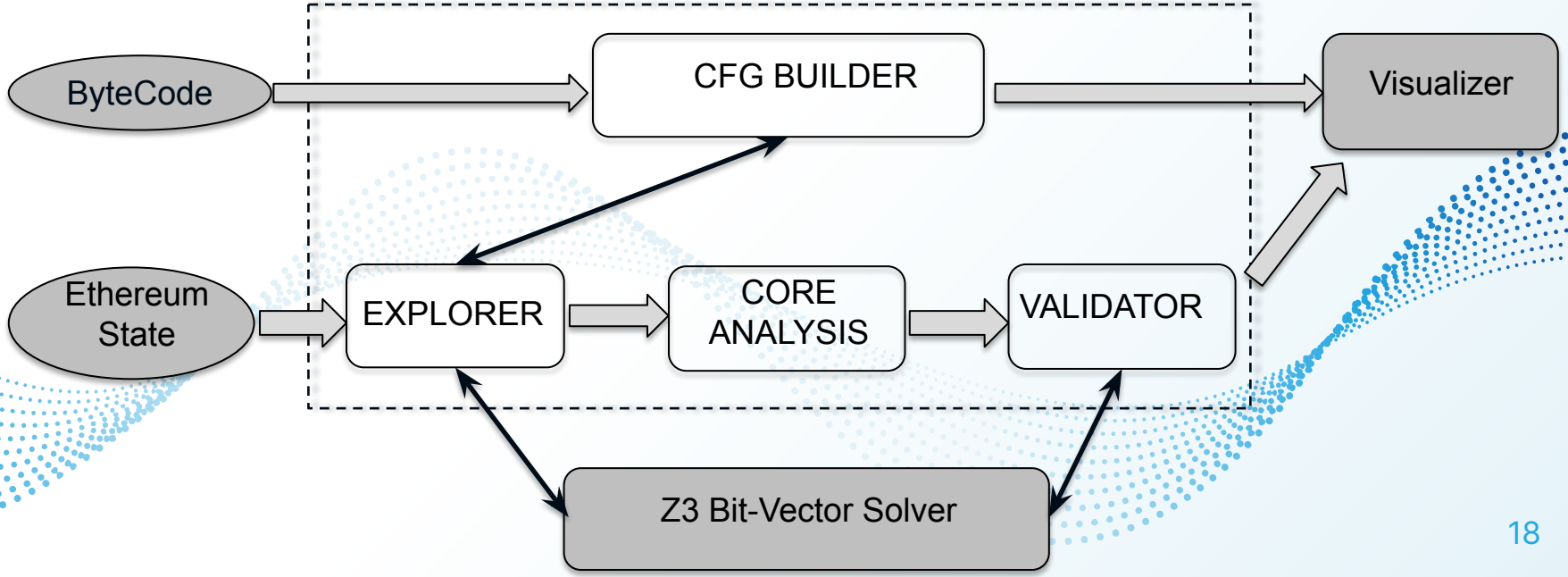
- Miners can vary the block timestamp

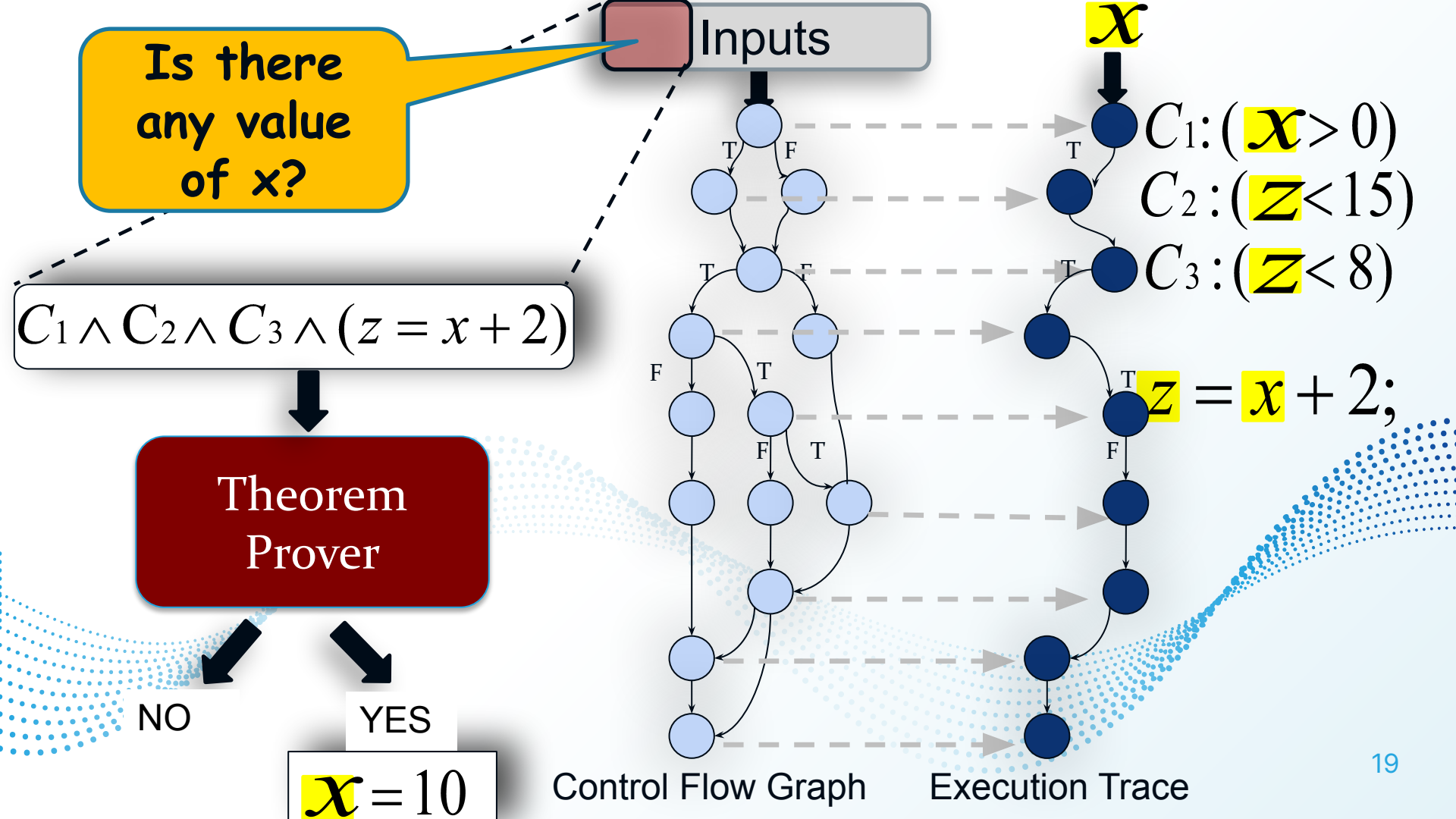
```
block.timestamp <= now + 900 &&
block.timestamp >= parent.timestamp
```

- Bias the output of contract execution to their benefit
  - Timed puzzles, time-based RNGs

Oyente Architecture

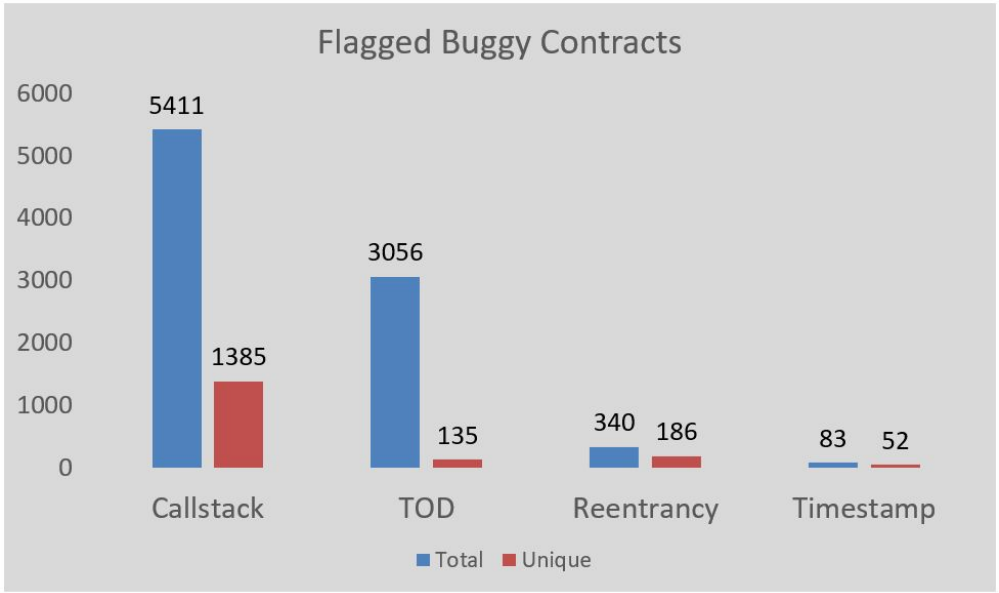
- Based on symbolic execution
- Modularized architecture




Is there any value of x?
Inputs
$C_1 \wedge C_2 \wedge C_3 \wedge (z = x + 2)$
Theorem Prover
NO
YES
$x = 10$
T
F
x
$C_1 : (x > 0)$
$C_2 : (z < 15)$
$C_3 : (z < 8)$
$z = x + 2;$
Control Flow Graph
Execution Trace

Evaluation

- Detect buys in real-world smart contracts
  - Run with 19,366 contracts
  - 30 mins timeout



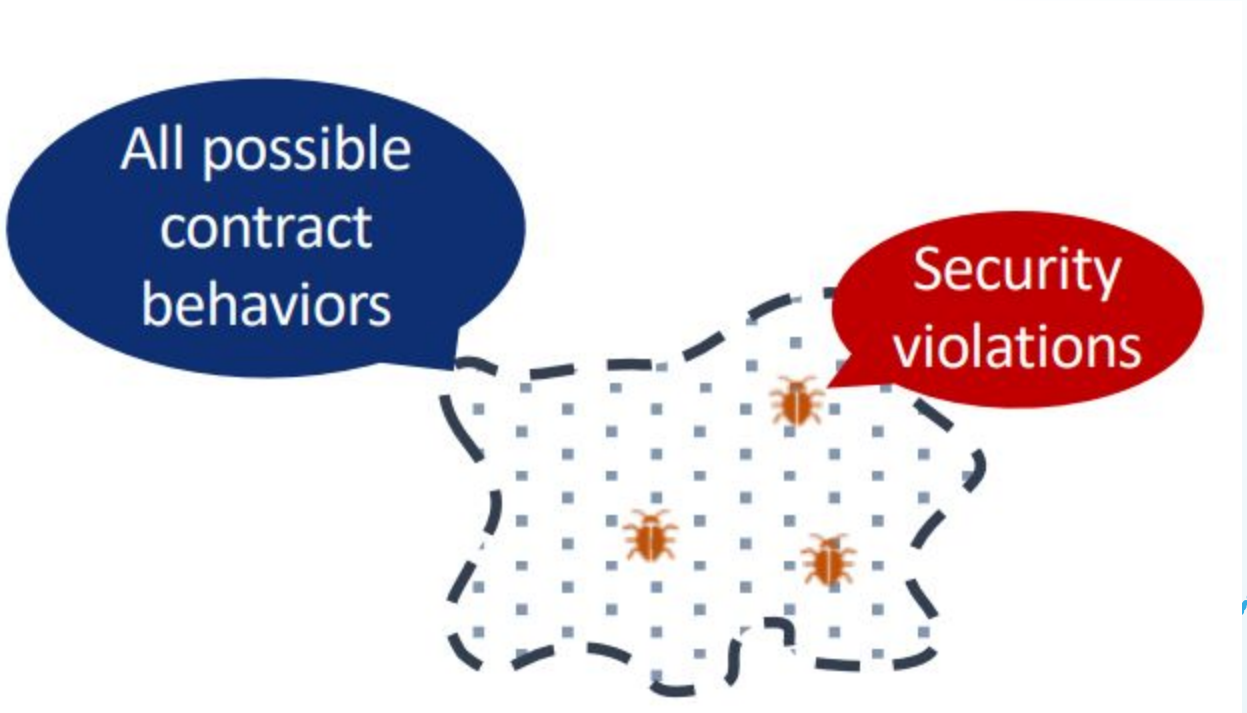
Securify: Practical Security Analysis of Smart Contracts

Petar Tsankov, Andrei Dan Dana, Drachsler-Cohen, Arthur Gervais,

Florian Bünzli, Martin Vechev

ACM CCS 2018

Motivation



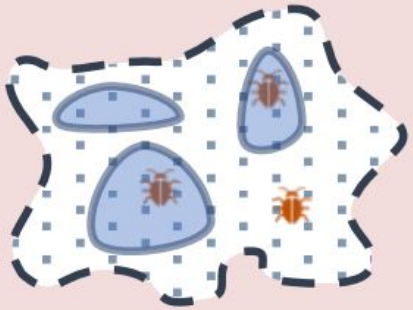
Motivation

Truffle

Testing

Report true bugs
Can miss bugs

Oyente, Mythril, MAIAN

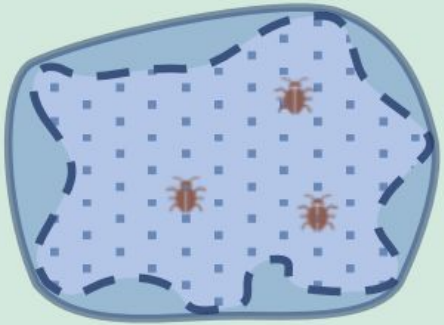


Dynamic (symbolic) analysis

Report true bugs
Can miss bugs

Bug finding

WANTED: Automated Verifier



Can report false alarms
No missed bugs

Verification

Key insight

- When contracts satisfy/violate a **property**, they often also satisfy/violate **a much simpler property**

Security property
No state changes after call instructions

Hard to verify in general

```
function withdraw() {
  uint amount = balances[msg.sender];
  msg.sender.call.value(amount)();
  balances[msg.sender] = 0;
}
```

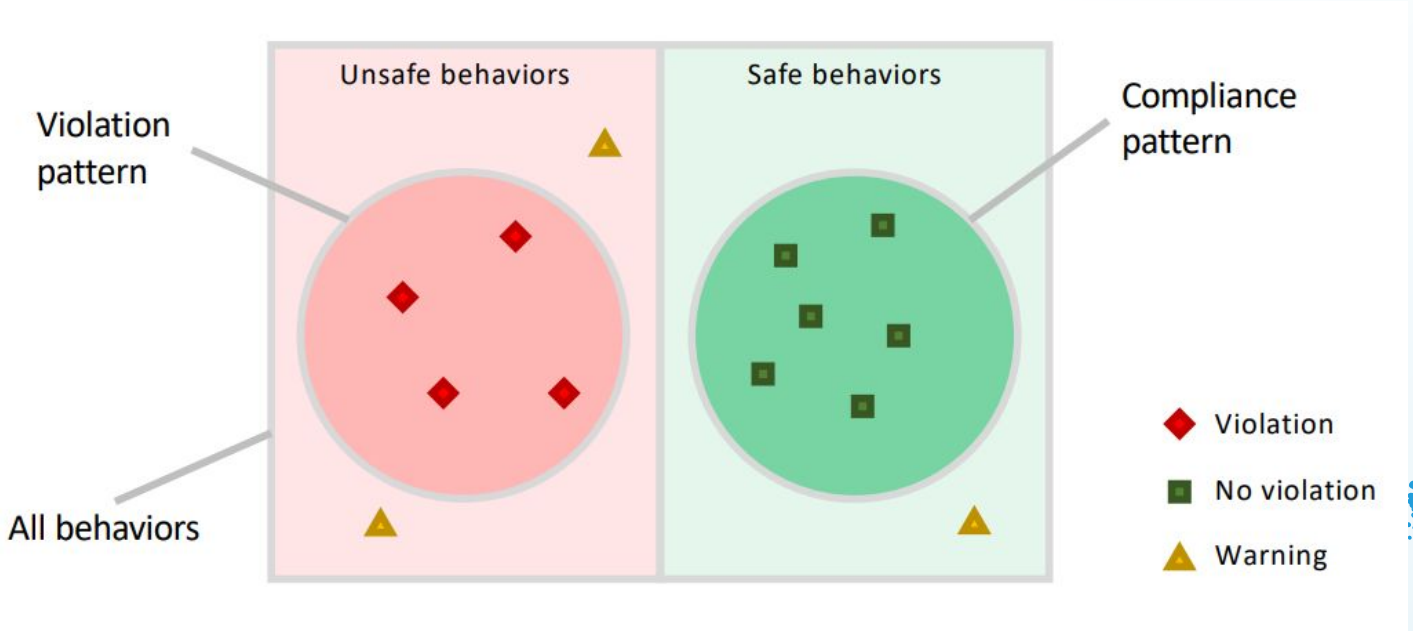
Compliance pattern
No writes to storage **may follow** call instructions

Verifies 91% of all deployed contracts

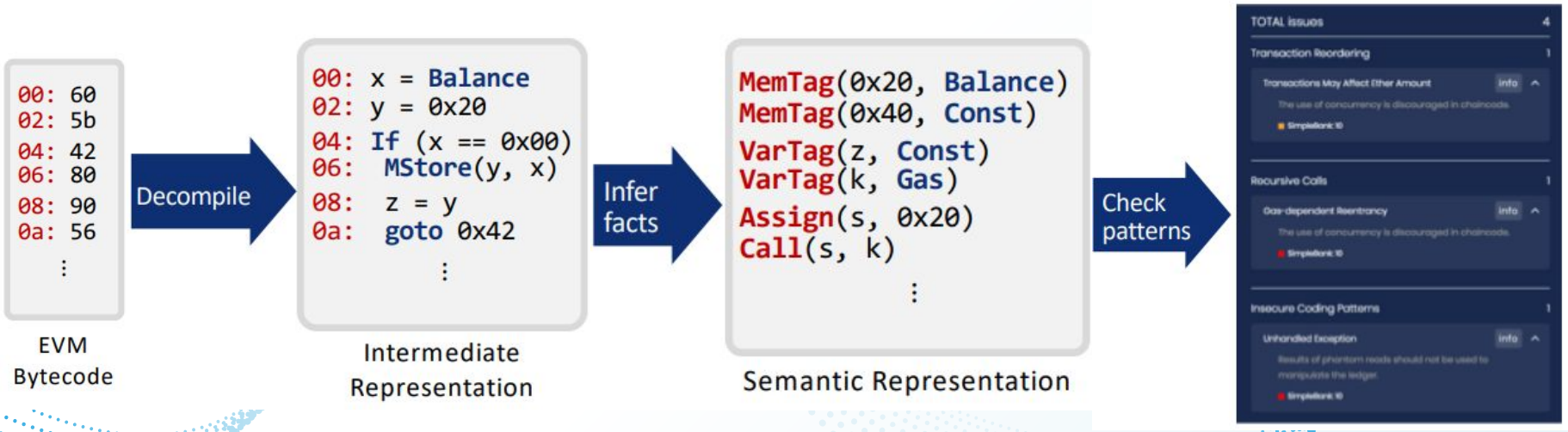
Violation pattern
A write to storage **must follow** call instructions

Easier to check automatically

Classifying Behaviors

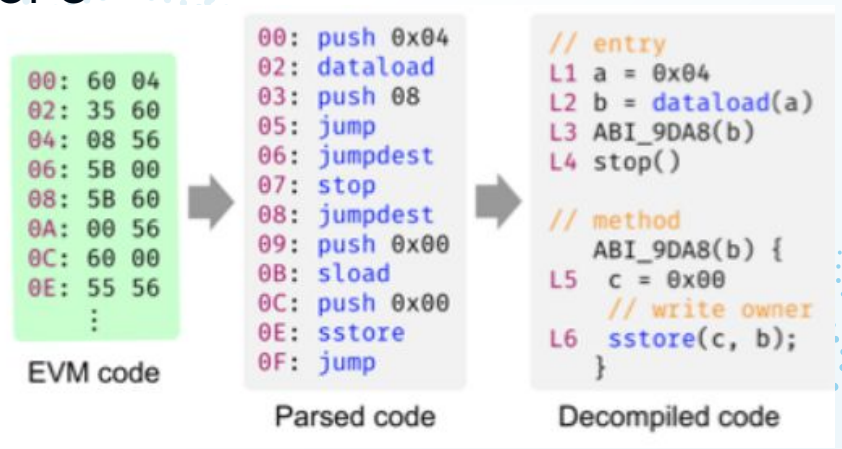


Securify: A Practical Verifier



From EVM to CFG over SSA

- Decompiling EVM bytecode
  - Convert into SSA format
  - Perform partial evaluation
    - Resolve jump dest, memory/storage offsets
  - Identify and inline methods
  - Construct CFG



Facts Inference

- Many properties can be checked on the contracts' dependency graph

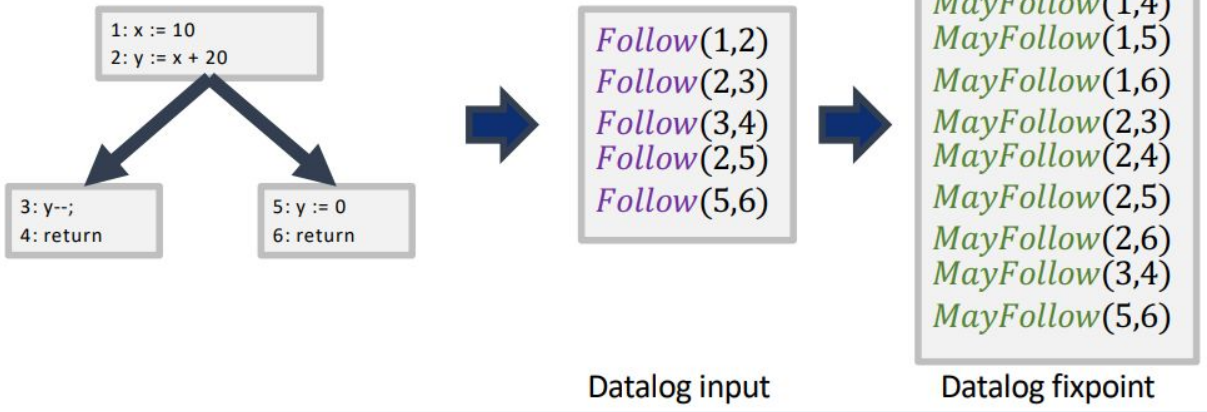
Flow dependencies	
$MayFollow(l, l')$	The instruction at label l may follow that at label l'
$MustFollow(l, l')$	The instruction at label l must follow that at label l'
Data dependencies	
$MayDepOn(x, t)$	The value of x may depend on tag t
$DetBy(x, t)$	For different values of t the value of x is different.

A tag can be an instruction (e.g. Caller) or a variable

Facts Inference

- Example: *MayFollow*

$$MayFollow(i,j) \leftarrow Follow(i,j)$$
$$MayFollow(i,j) \leftarrow Follow(i,k), MayFollow(k,j)$$



Deriving MayDepOn

```
1: x := Balance
2: Mstore(0x20, x)
3: y := MLoad(0x20)
4: z := x + y
```

$Follow(1,2)$
$Follow(2,3)$
$Follow(3,4)$
$Assign(x, Balance)$
$IsConst(0x20)$
$MStore(2, 0x20, x)$
$MLoad(3, y, 0x20)$
$Op(4, z, x)$
$Op(4, z, y)$

Derived from the Balance instruction

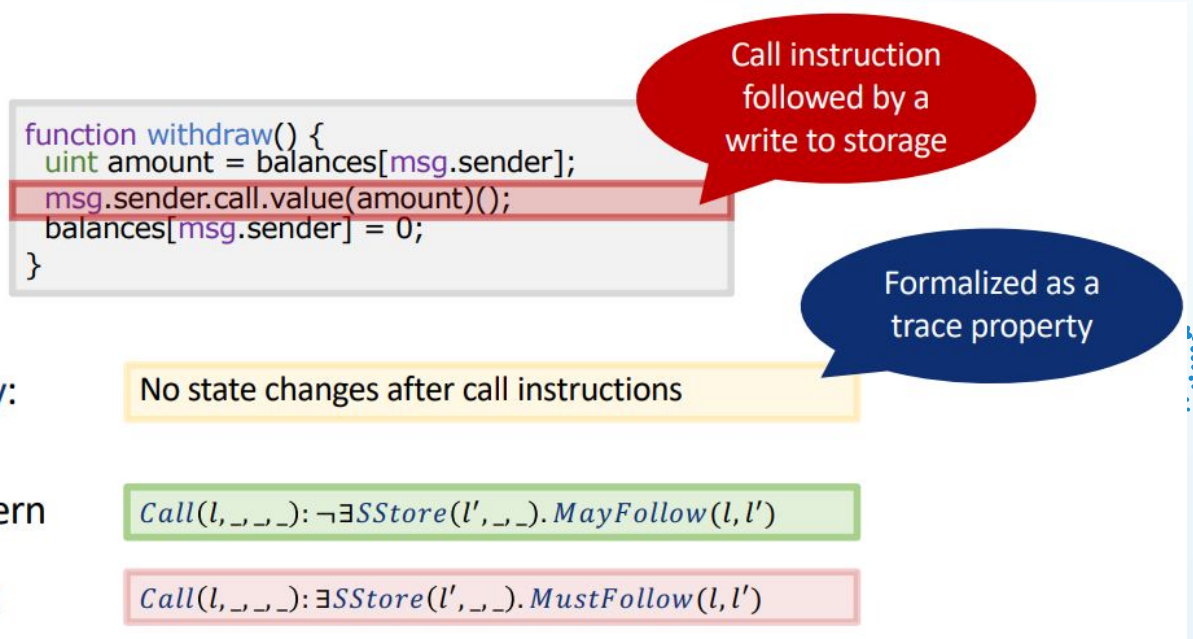
Memory operations

Capture that z is derived from x and y

$$MayDepOn(x,t) \leftarrow Assign(x,t)$$
$$MayDepOn(x,t) \leftarrow Op(\_,x,x'), MayDepOn(x',t)$$
$$MayDepOn(x,t) \leftarrow MLoad(l,x,o), isConst(l,o), MemTag(l,o,t)$$
$$MayDepOn(x,t) \leftarrow MLoad(l,x,o), \neg isConst(l,o), MemTag(l,\_,t)$$

$$MemTag(l,o,t) \leftarrow MStore(l,o,x), isConst(o), MayDepOn(x,t)$$
$$MemTag(l,\top,t) \leftarrow MStore(l,o,x), \neg isConst(o), MayDepOn(x,t)$$
$$MemTag(l,o,t) \leftarrow Follows(l,l'), MemTag(l',o,t), \neg MStore(l,o,\_)$$

Pattern Check: the DAO attack



Pattern Check: Unrestricted Writes

```
address owner = ...;

function initWallet(address _owner) {
  owner = _owner;
}
```

Unrestricted write

Formalized as a hyperproperty

Security property: No storage offset is writable by all users

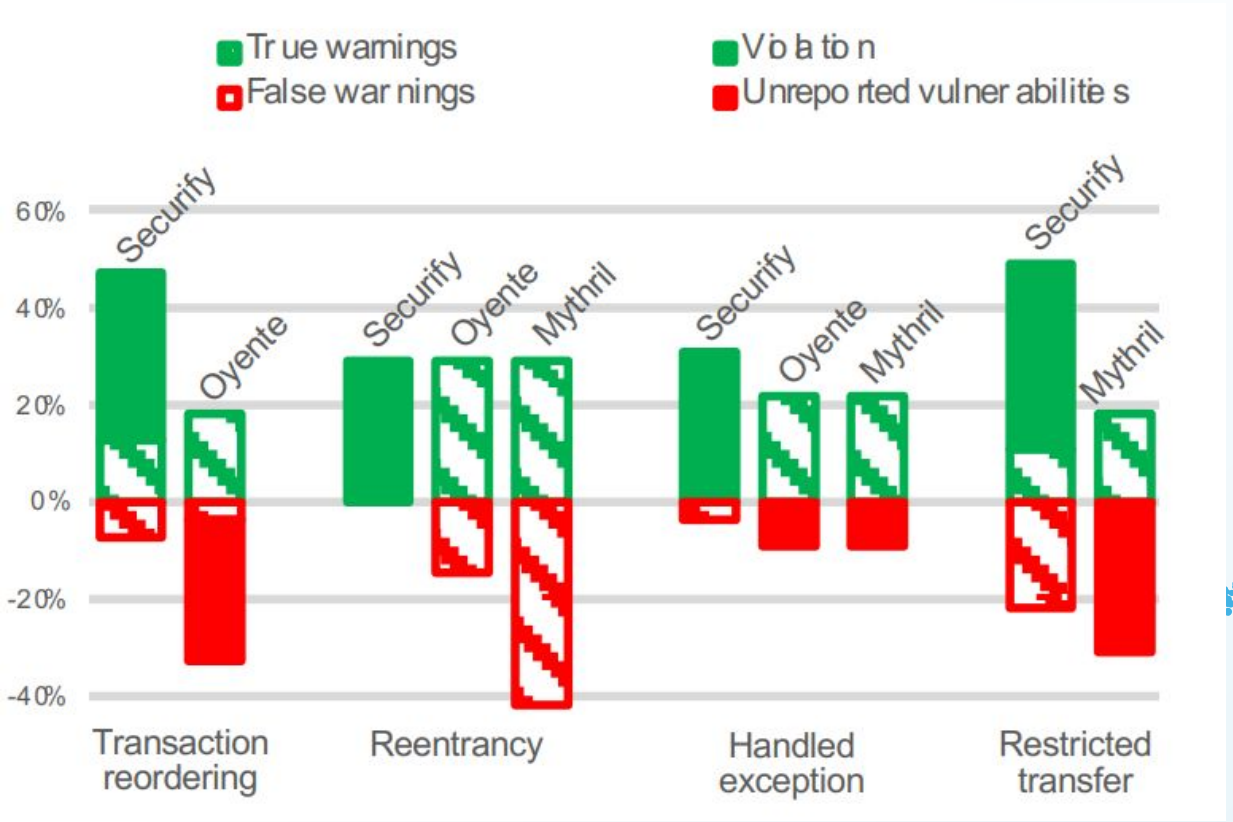
Compliance pattern: $SStore(\_, x, \_): DetBy(x, Caller)$

Violation pattern: $SStore(l, x, \_): \neg MayDepOn(x, Caller) \wedge \neg MayDepOn(l, Caller)$

Evaluation

- Dataset
  - 80 open-source smart contracts
- Baseline
  - Oyente
  - Mythril
- Experiment
  - Run contracts using the three tools
  - Manually inspect each reported vulnerability

Evaluation


True warnings
False warnings
Violation
Unreported vulnerabilities
60%
40%
20%
0%
-20%
-40%
Securify
Oyente
Securify
Oyente
Mythril
Securify
Oyente
Mythril
Securify
Mythril
Transaction reordering
Reentrancy
Handled exception
Restricted transfer

Thank you!

Questions?