

Blockchain Security Smart Contract Analysis

Yue Duan

Outline

-
- Research paper:
 - Making Smart Contracts Smarter
 - Security: 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





I think TheDAO is getting drained right now

89d • ledgerwatch • self.ethereum



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 DApp (a "contract"), living on the blockchain, that will make you a King or Queen, might grant you riches, and will immortalize your name.



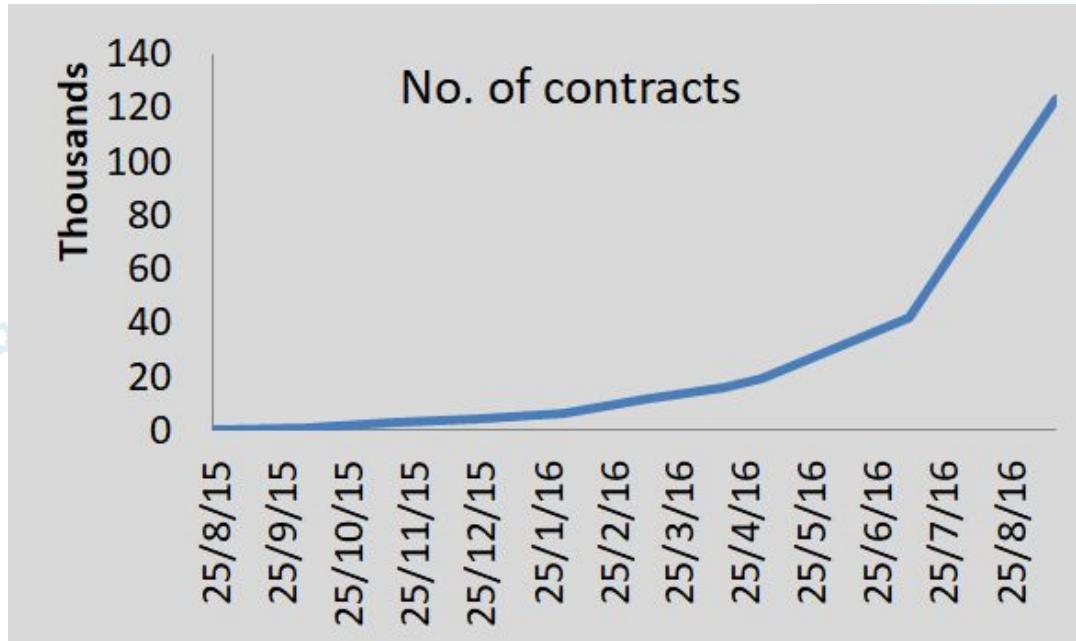
Important Notice

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 }
```



```
60606040526040516102503  
80380610250833981016040  
528.....
```



```
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

Anyone can submit a solution to claim the reward

Owner can update the reward anytime

PuzzleSolver Contract

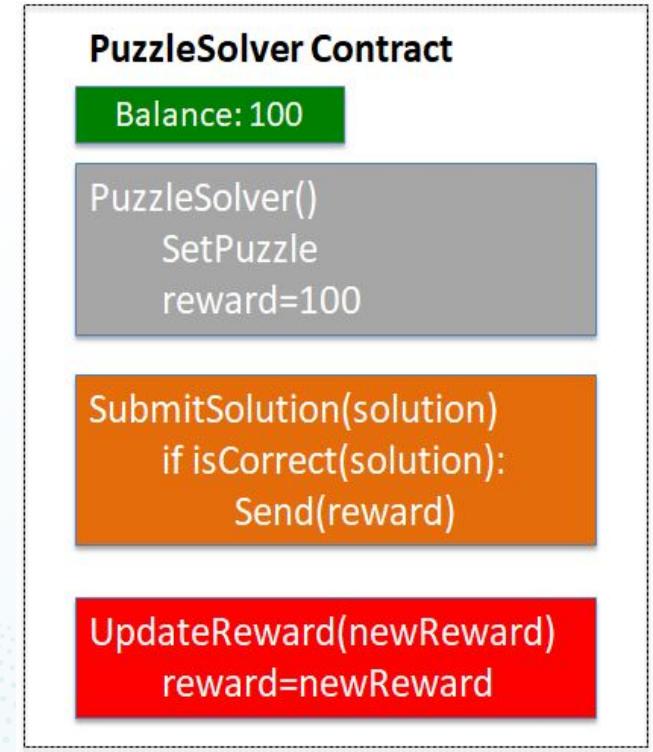
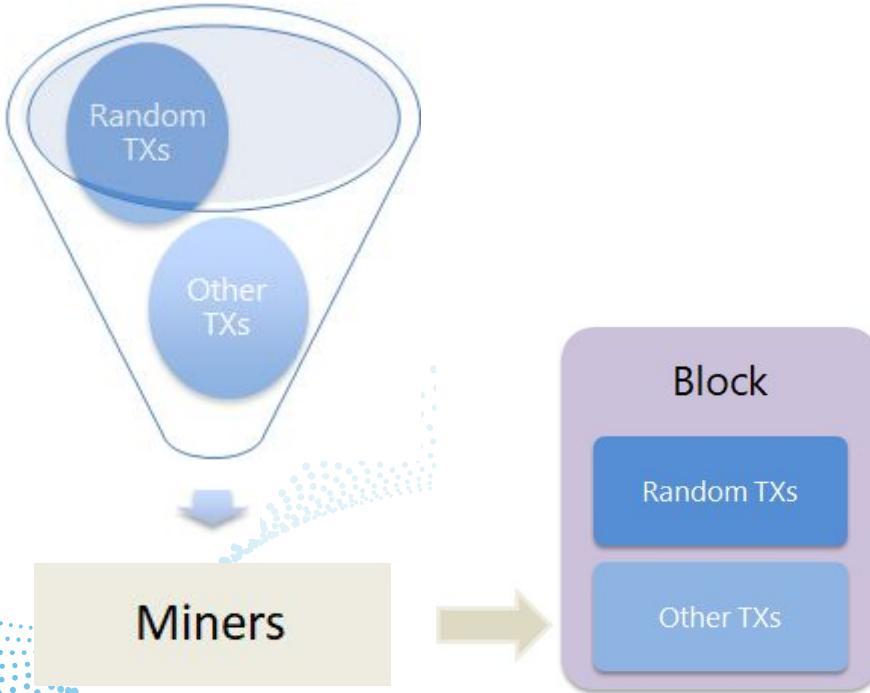
Balance: 100

PuzzleSolver()
SetPuzzle
reward=100

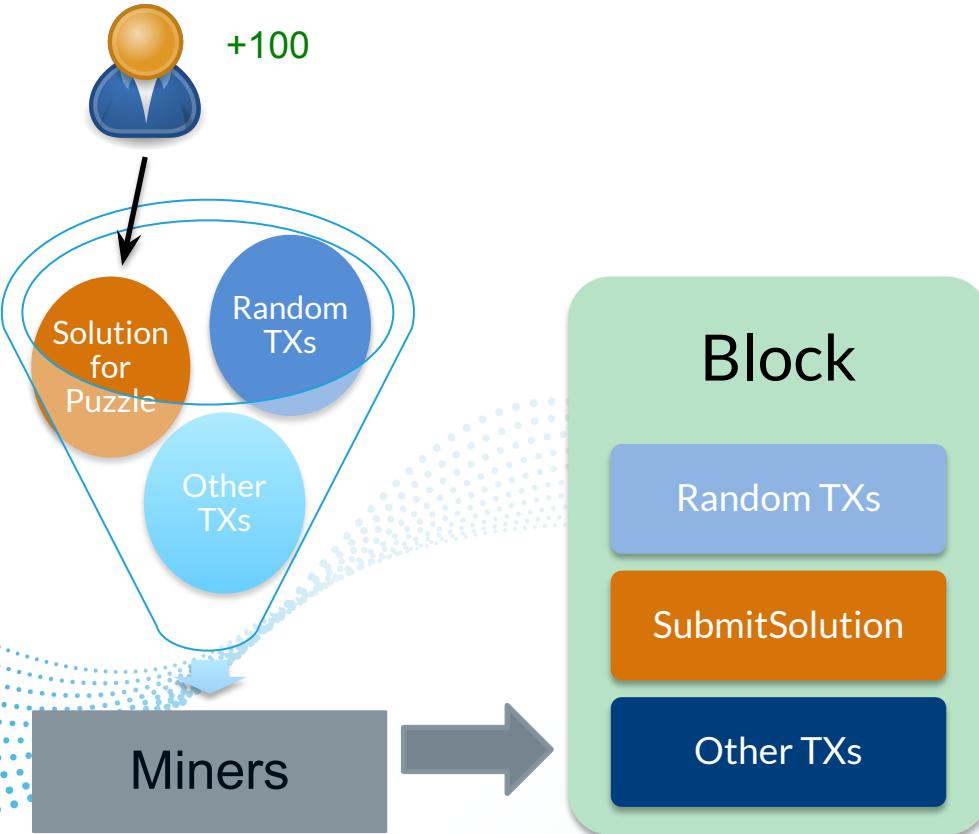
SubmitSolution(solution)
if isCorrect(solution):
Send(reward)

UpdateReward(newReward)
reward=newReward

Scenario 1



Scenario 2



PuzzleSolver Contract

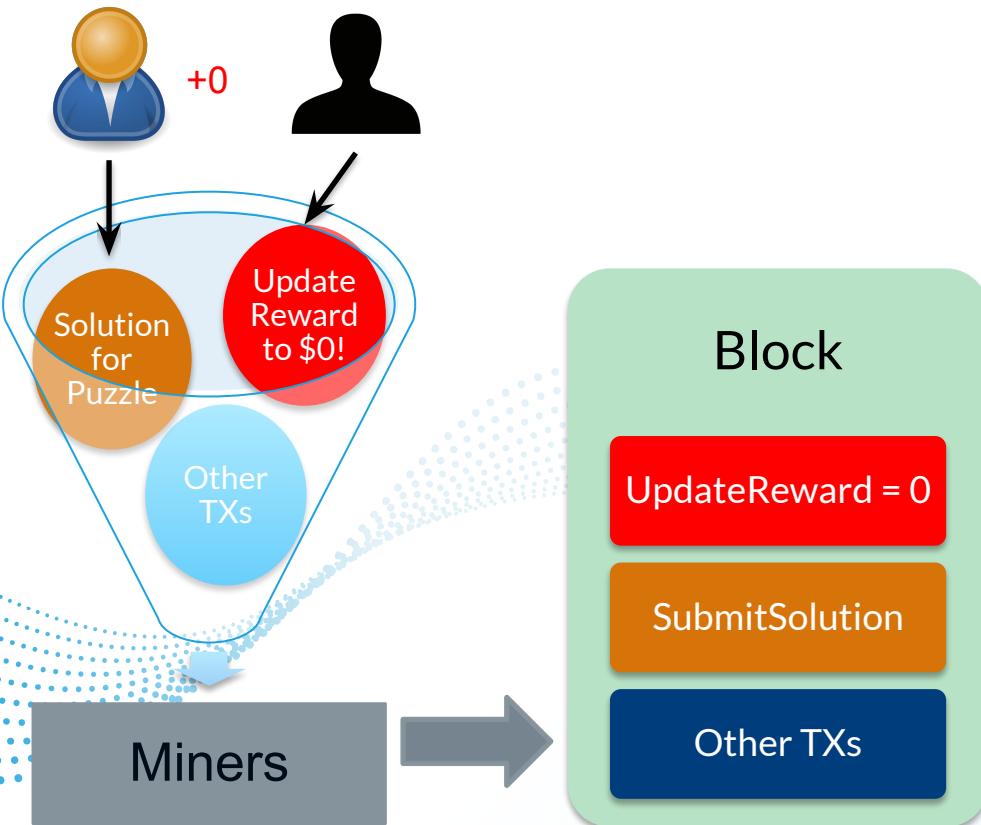
Balance: 0

PuzzleSolver()
SetDifficulty
reward=100

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

UpdateReward(newReward)
reward=newReward

Scenario 3



PuzzleSolver Contract

Balance: 0

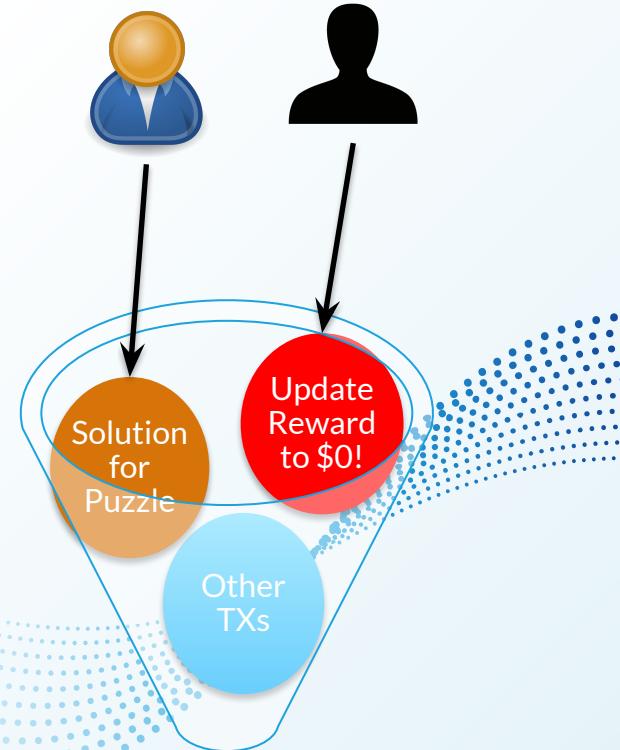
PuzzleSolver()
SetDifficulty
reward=100

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

UpdateReward(newReward)
reward=newReward

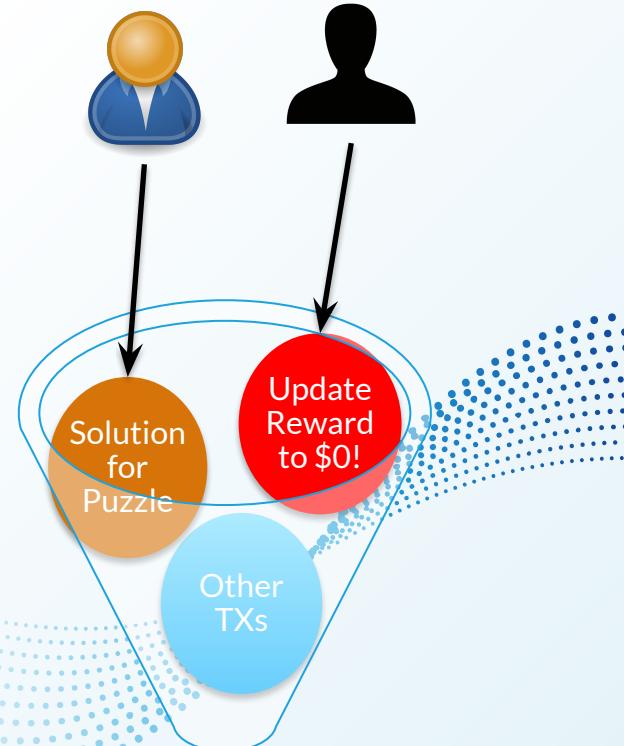
Transaction Ordering Dependence

- Observed state \neq 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

randomness =
F(timestamp)

```
1 contract theRun {
2     uint private LastPayout = 0;
3     uint256 salt = block.timestamp;
4     function random() returns (uint256 result){
5         uint256 y = salt * block.number/(salt%5);
6         uint256 seed = block.number/3 + (salt%300)
7             + LastPayout +y;
8
9         //h = the blockhash of the seed-th last block
10        uint256 h = uint256(block.blockhash(seed));
11
12        //random number between 1 and 100
13        return uint256(h % 100) + 1;
14    }
15    ...
16 }
```

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 jackpot to the last creditor
8         creditorAddresses[nCreditors - 1]
9             .send(profitFromCrash);
10        owner.send(this.balance);
11        ...
12    }
13 }
```

Timestamp is not reliable

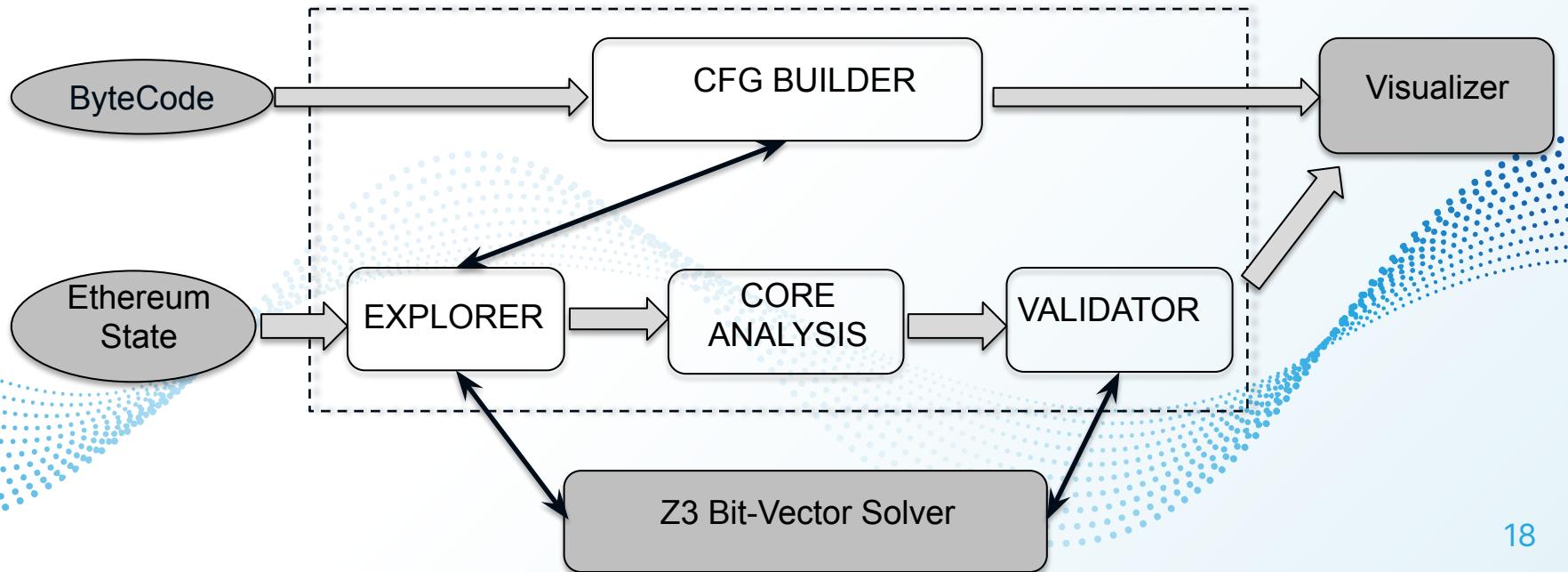
- Miners can vary the block timestamp

```
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

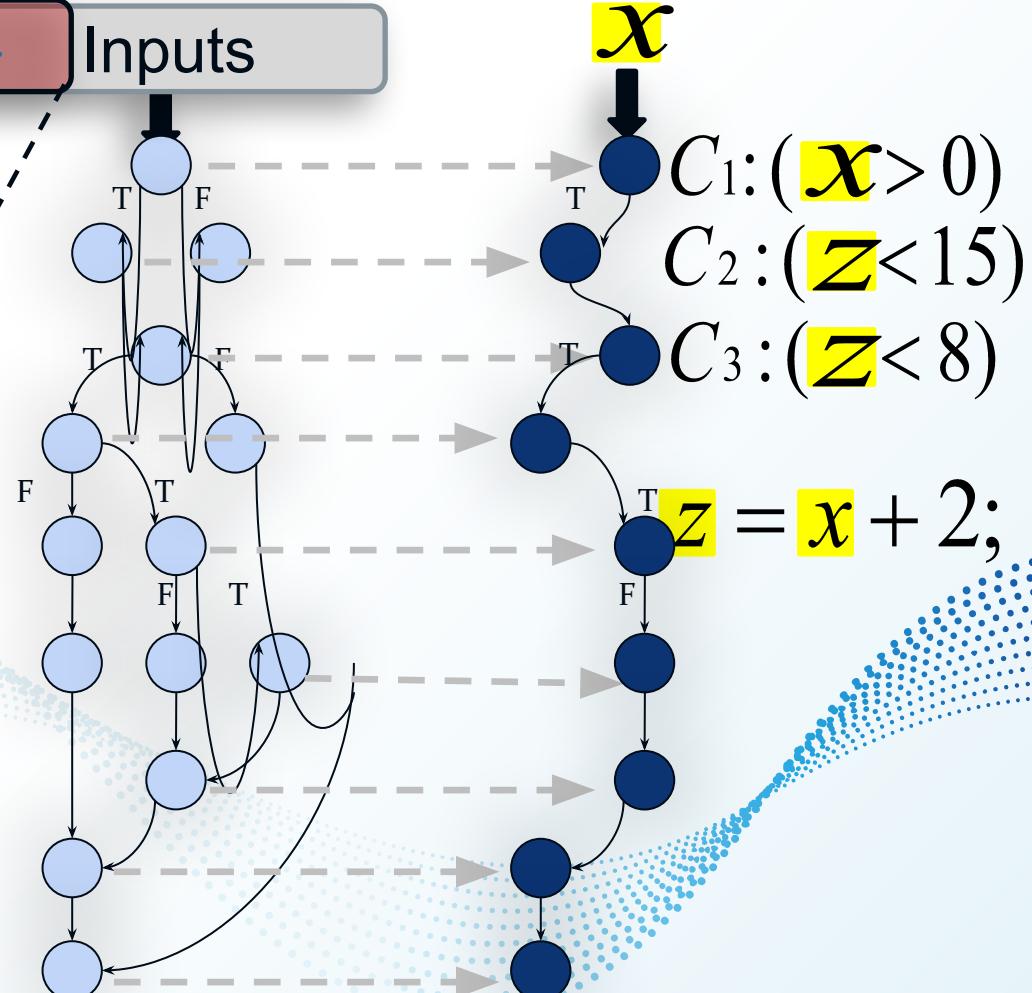
NO

YES

$$x = 10$$

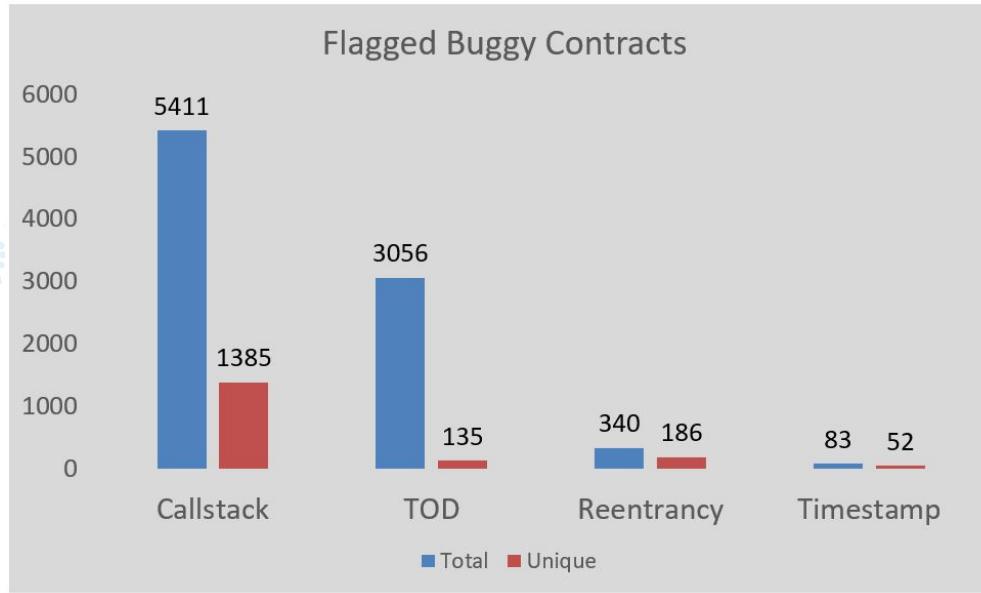
Control Flow Graph

Execution Trace



Evaluation

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



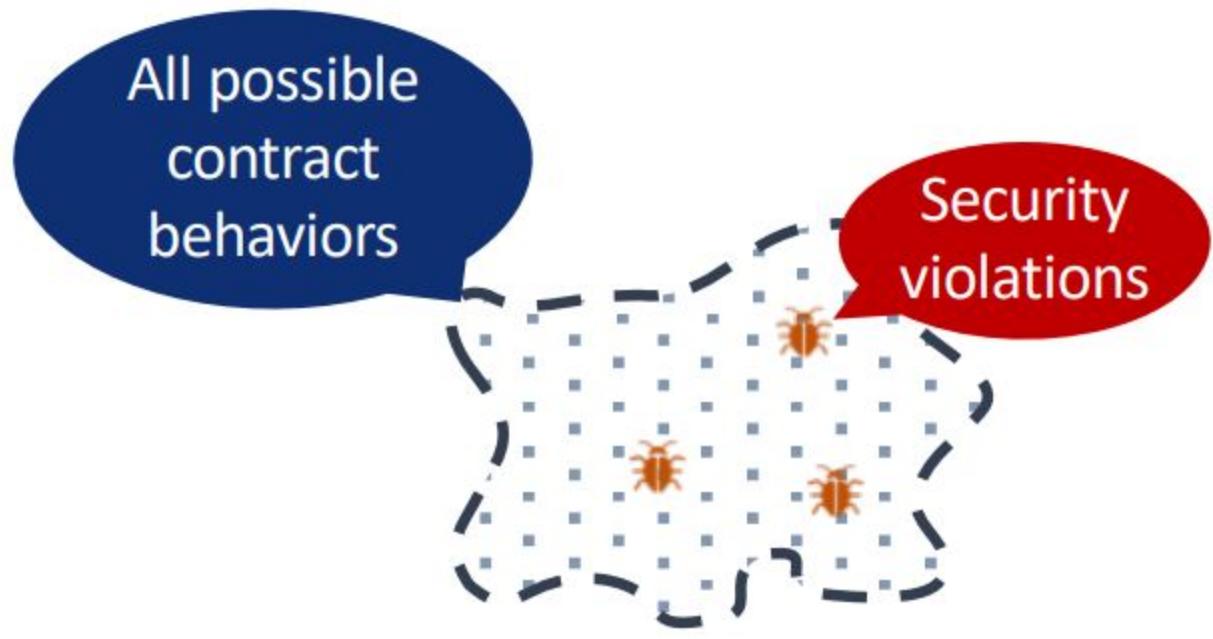
Securify: Practical Security Analysis of Smart Contracts

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

Florian Bünzli, Martin Vechev

ACM CCS 2018

Motivation



Motivation

Truffle

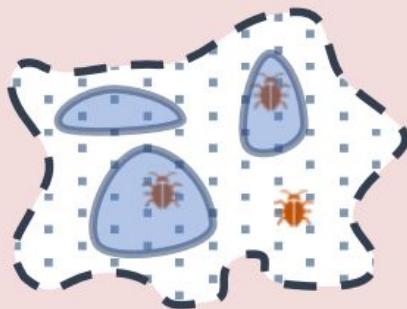


Testing

Report true bugs
Can miss bugs

Bug finding

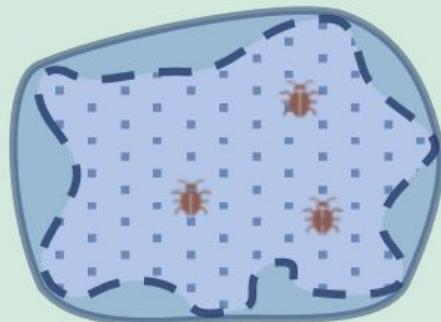
Oyente, Mythril, MAIAN



Dynamic (symbolic) analysis

Report true bugs
Can miss bugs

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

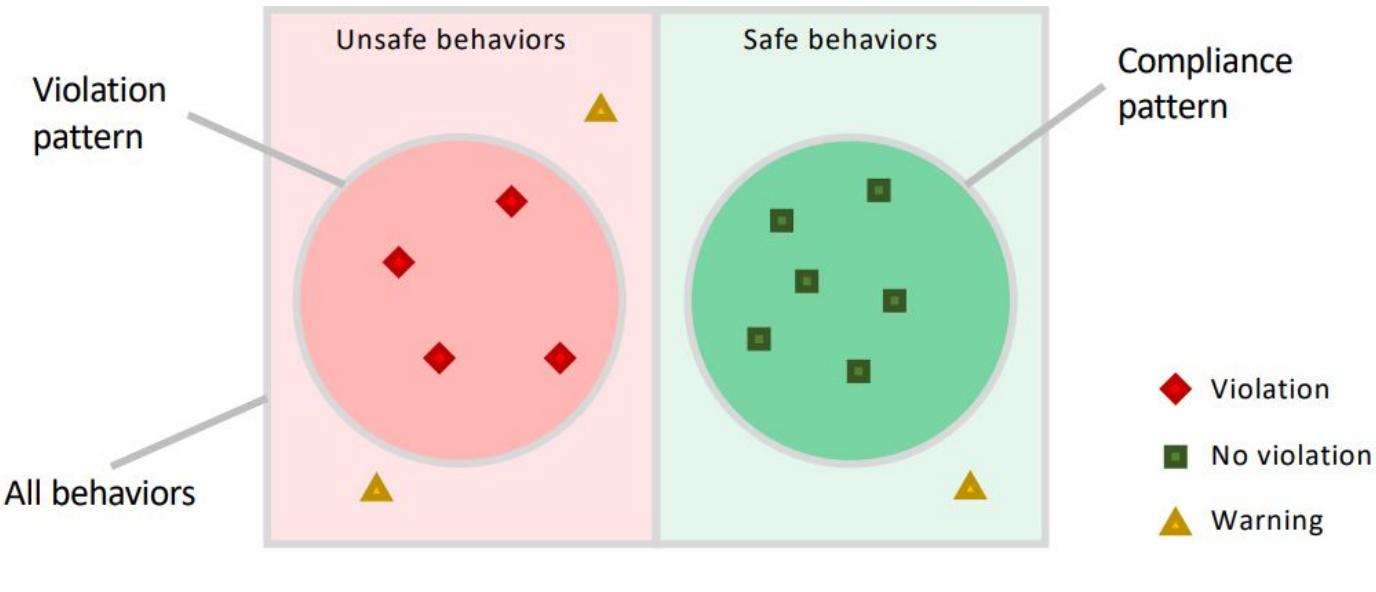
Violation pattern

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

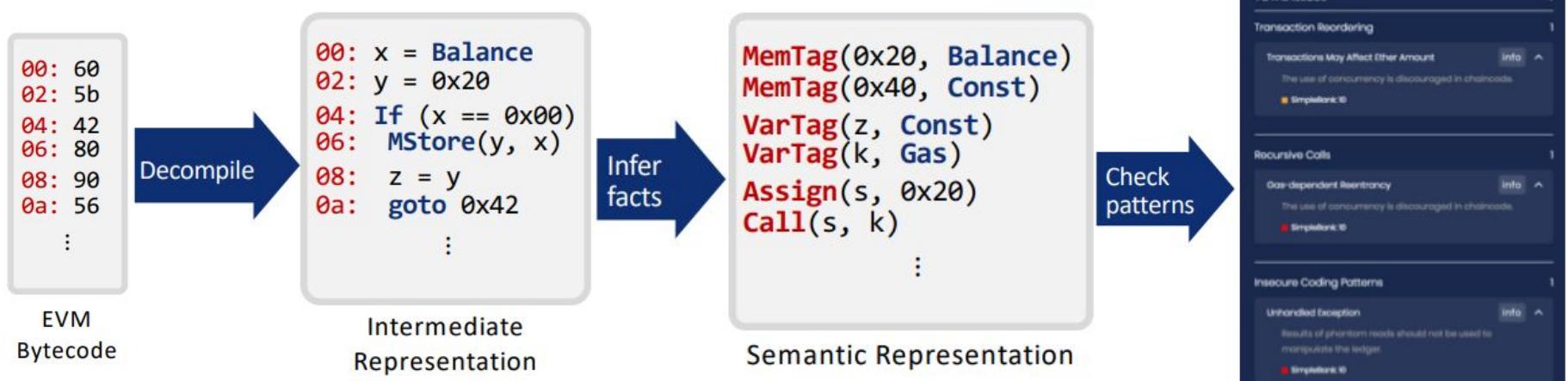
Verifies 91% of all
deployed contracts

Easier to check
automatically

Classifying Behaviors

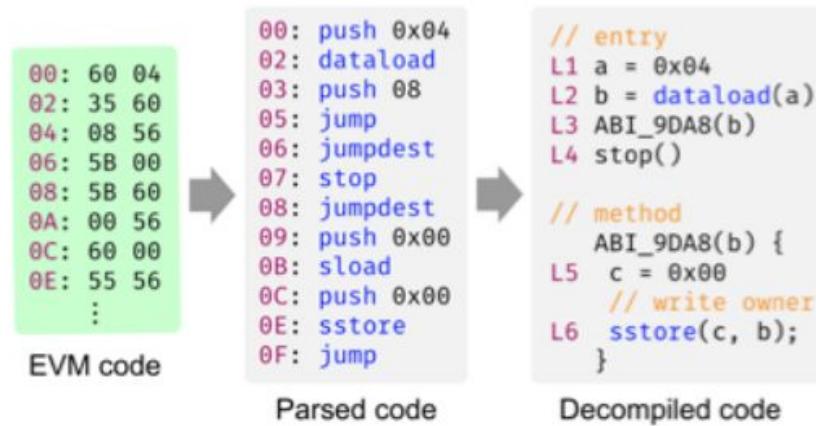


Securify: A Practical Verifier



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

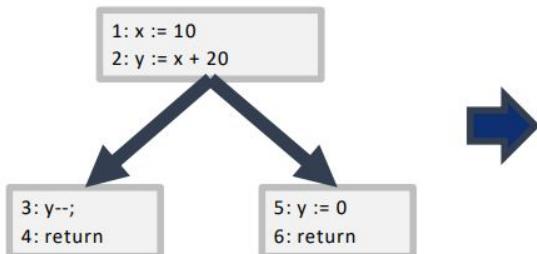
Flow dependencies	
$MayFollow(l, l')$	The instruction at label l may follow that at label l'
$MustFollow(l, l')$	The instruction at label l must follow that at label l'
Data dependencies	
$MayDepOn(x, t)$	The value of x may depend on tag t
$DetBy(x, t)$	For different values of t the value of x is different.

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

Facts Inference

- Example: *MayFollow*

```
MayFollow(i,j) ← Follow(i,j)  
MayFollow(i,j) ← Follow(i,k), MayFollow(k,j)
```



Datalog input

```
Follow(1,2)  
Follow(2,3)  
Follow(3,4)  
Follow(2,5)  
Follow(5,6)
```

Datalog fixpoint

```
MayFollow(1,2)  
MayFollow(1,3)  
MayFollow(1,4)  
MayFollow(1,5)  
MayFollow(1,6)  
MayFollow(2,3)  
MayFollow(2,4)  
MayFollow(2,5)  
MayFollow(2,6)  
MayFollow(3,4)  
MayFollow(5,6)
```

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

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

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

Pattern Check: the DAO attack

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

Call instruction
followed by a
write to storage

Formalized as a
trace property

Security property: No state changes after call instructions

Compliance pattern $\text{Call}(l, _, _, _): \neg \exists S\text{Store}(l', _, _). \text{MayFollow}(l, l')$

Violation pattern $\text{Call}(l, _, _, _): \exists S\text{Store}(l', _, _). \text{MustFollow}(l, l')$

Pattern Check: Unrestricted Writes

Security property:

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

Unrestricted write

Formalized as a hyperproperty

No storage offset is writable by all users

Compliance pattern

$SStore(_, x, _): DetBy(x, Caller)$

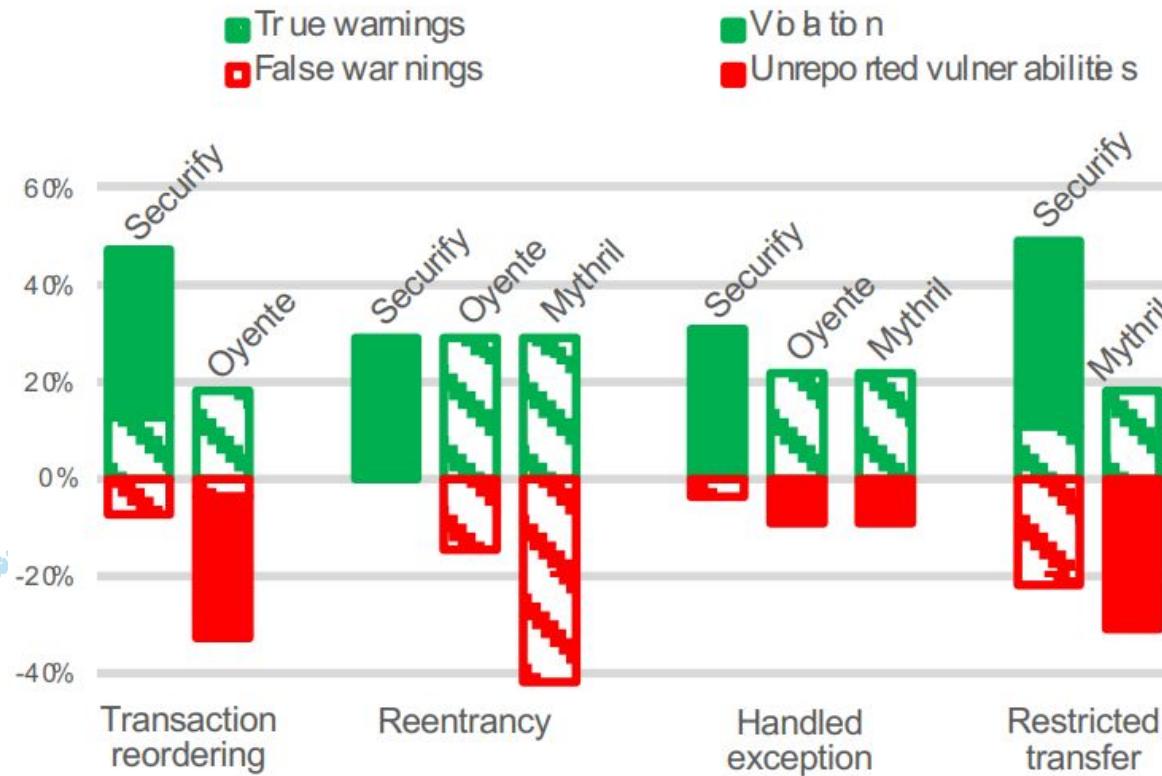
Violation pattern

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

Evaluation

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

Evaluation



Thank you!

Questions?

