A Federated Society of Bots for Smart Contract Testing

Mariano Ceccato

University of Verona

mariano.Ceccato@univr.it

In collaboration with:

Emanuele Viglianisi – Fondazione Bruno Kessler

Paolo Tonella – Università della Svizzera Italiana USI

De Cifris seminar, University of Trento

January 23rd, 2020

From cryptocurrencies to smart contracts

- Bitcoin: the state of an account with a given address holds some coins (balance)

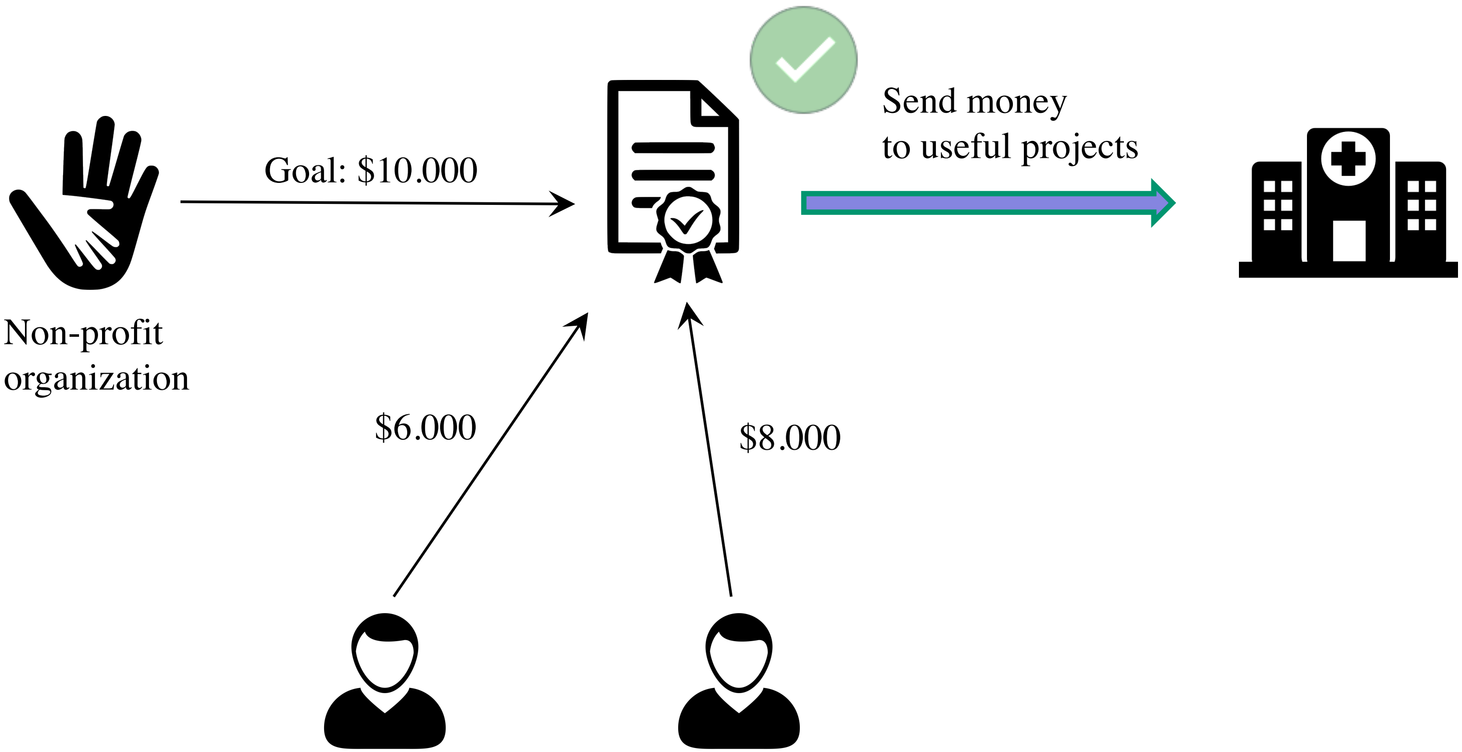
- Ethereum: accounts include coins, executable code and persistent (private) storage (balance, code, storage)

- **Smart contract:** full-fledged program that is run on a blockchain and implements a contract between users
  – saving wallets, investments, insurances, games, etc.
  – +80.2% annual growth rate

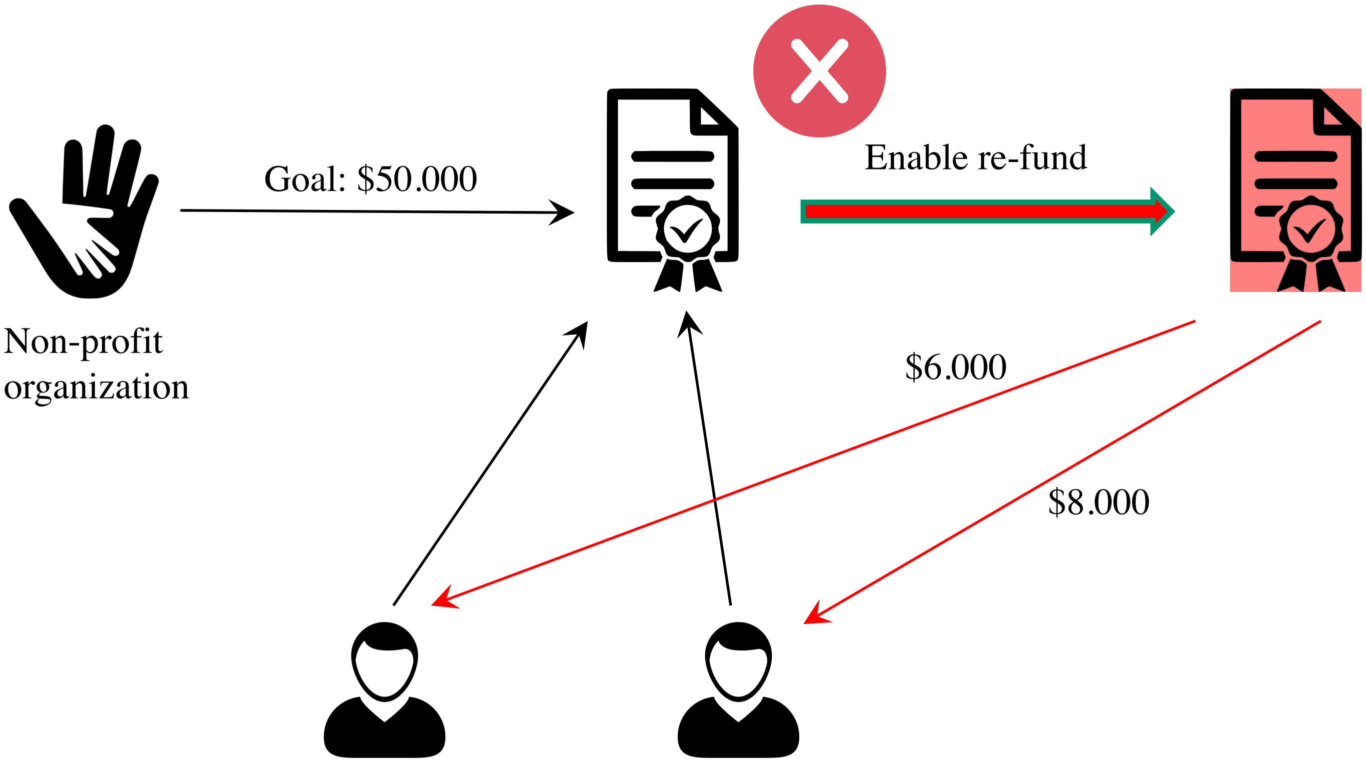
Example:
Smart contract for charity

- A new un-known non-profit organization want to run a crowdfunding campaign to start a new charity project.
- The organization fixes a goal, and wants the contributors to be able to ask for a re-fund if the goal is not reached.

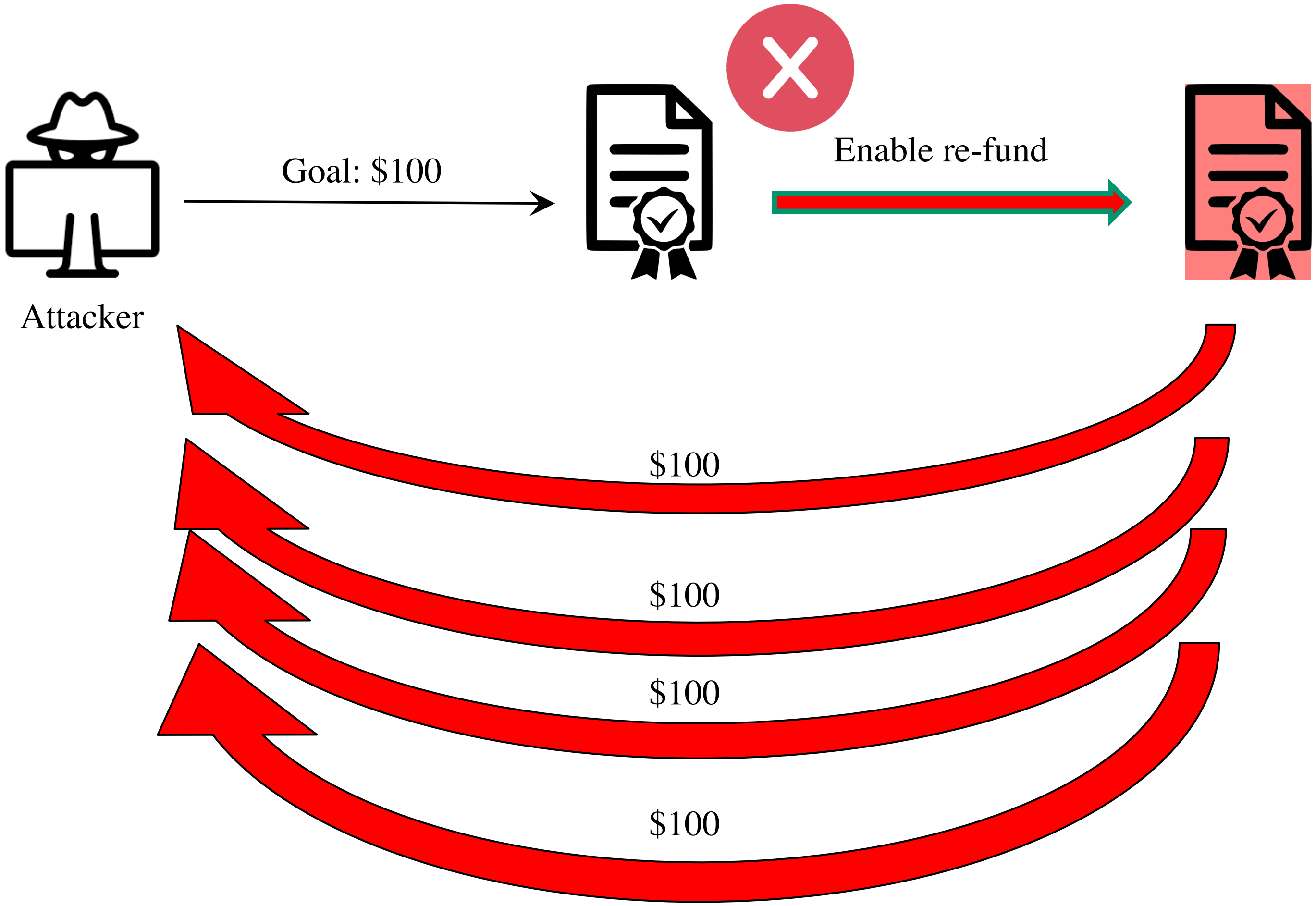
Successful campaign



Unsuccessful campaign



Attack



A prominent case

- DAO: Decentralized Autonomous Organization
- Decentralized venture capital
- Receive Ether for DAO tokens
- $150M

- June 17, 2016 a loophole was found in the smart contact
- A hacker could steal $70M in DAO tokens
- Ethereum hard fork required

Arithmetic Overflow

- Limited size to represents number

- Increment on the max value 999,999 resets the counter to 0.

Multiple Roles

```
function buggedTransferFrom(
  address _from,
  address _to,
  uint256 _value
)
  public
  returns (bool)
{
  require(_value <= allowed[_from][msg.sender]);
  require(msg.sender != _from && _from != _to);

  balances[_from] -= _value;
  balances[_to] += _value;
  allowed[_from][msg.sender] -= _value;

  emit Transfer(_from, _to, _value);
  return true;
}
```

Spender

Initiator

Receiver

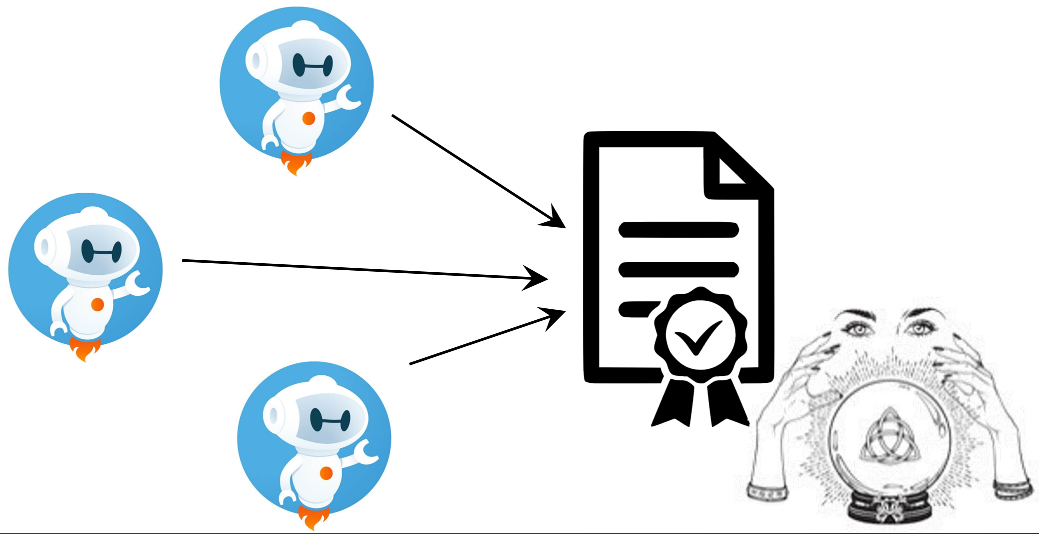
Problem definition

- Programs have bugs, but bugs in smart contracts might generate illegal gains and losses
- Bugs in smart contracts cannot be patched (blockchain transactions are irreversible)

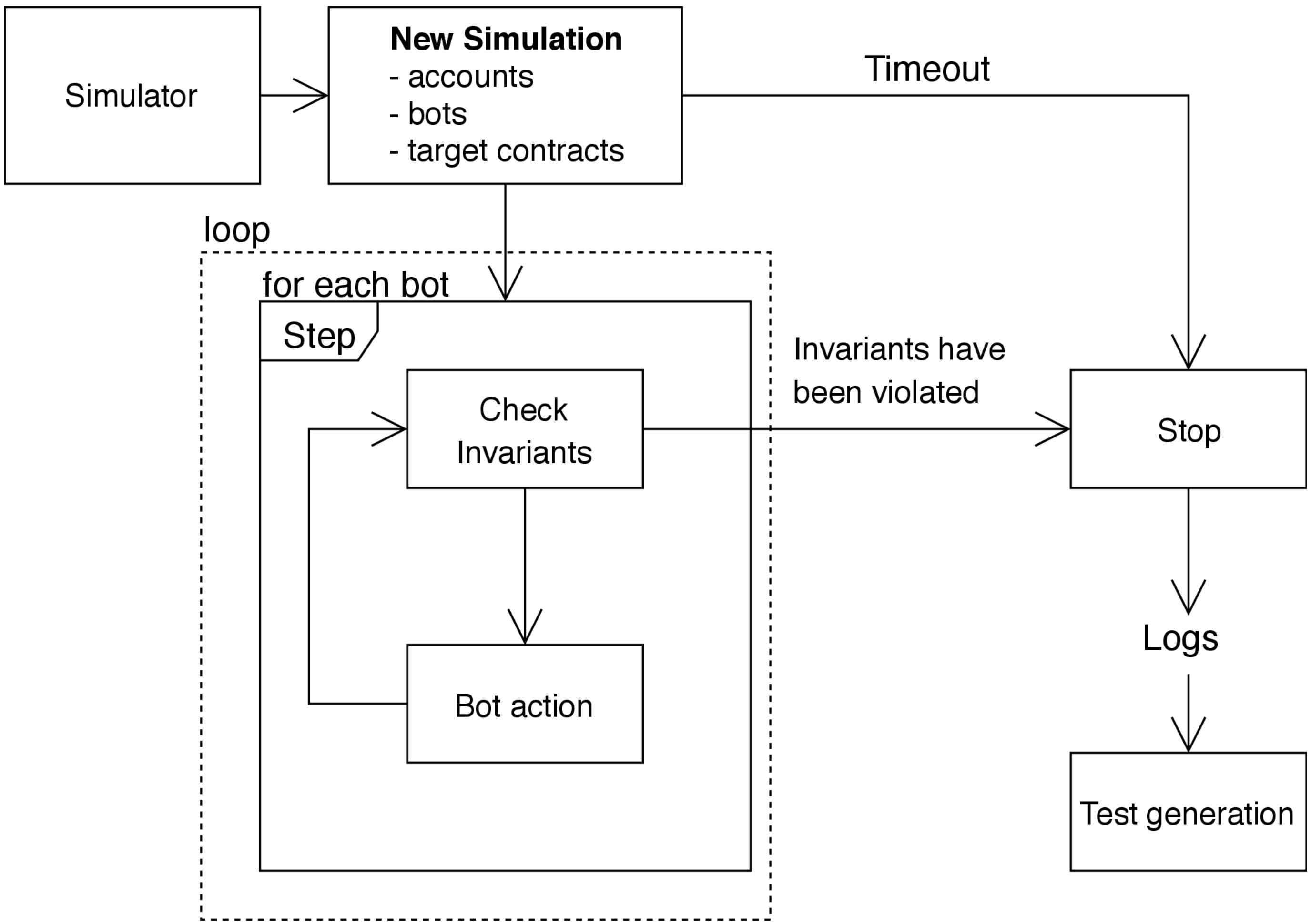
- <u>Objective</u>: detect programming errors, before erroneous transactions are permanently sorted in the blockchain

Novel Contribution

- Testing-based approach to identify implementation mistakes in smart contracts
  - Automated input generation to execute the contract
  - Automated oracle to detect invalid states at testing time
  - Society of bots to test a smart contract with different interactions



SoCRATES



Simulator
New Simulation
- accounts
- bots
- target contracts
Timeout
loop
for each bot
Step
Check Invariants
Bot action
Invariants have been violated
Stop
Logs
Test generation

Random behavior

- Random values have high chance to violate initial *require* statements
  - *Uint* is 256-bit unsigned integer

- The test engineer should limit the range of values to those interesting for her contract
  - E.g., [0, *balance*+1000]

```
int      → random[min,max]
address  → random[accounts]
bool     → random[0,1]
string   → Random_string[min_length,max_lenght]
```

Boundary Behavior

- Parameter values near to the border of the parameter types
  - Type: uint256, delta=1000
  - Random value in [0,999] U [2^{256}-1000, 2^{256}-1]

Overflow behavior

```
function batchTransfer(address[] _receivers, uint256 _value) public returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i], _value);
    }
    return true;
}
```

- Static analysis: detection of arithmetic expressions that may overflow
  - [+, +=, ++, −, −=, −−, *, **, *=]
  - *safeMath* library not used
- Identification of **`require`** expressions

SMT Solver

```
function batchTransfer(address[] _receivers, uint256 _value) public returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i], _value);
    }
    return true;
}
```

```
// Returns overflow conditions for unsigned int
def is_overflow(value, numberOfBits):
    isAnOverflow = value > (2**numberOfBits) - 1
    isAnUnderflow = value < 0
    return Or(isAnOverflow, isAnUnderflow)

// inputs
cnt == input._receivers.length
amount == cnt * input._value

// force overflow condition on "amount"
is_overflow(amount, 256) == True

// requires
cnt > 0 AND cnt <= 20
_value > 0 AND
state.balances[sender] > amount
```

Oracle

- Invariants: conditions that should always be **`true`** in every contract state
- If an invariant is violated at testing time, the test exposed an implementation error

ID	Type	Invariant
I1	General	$\nexists t \in Txs$: successful(t) $\wedge$ overflow(t)
I2	EIP20	$\sum_{a \in accounts}$ balanceOf(a) = totalSupply
I3	EIP20	$\forall t \in Txs$: t=*transferFrom* $\Rightarrow$ t.amount $\leq$ $\text{allowance}_{\text{from,sender}}$
I4	EIP20	$\forall t \in Txs$: t=*transferFrom* $\Rightarrow$ allowance'=allowance−t.amount
I5	EIP20	$\forall t \in Txs$: t$\in${*transferFrom*,*transfer*} $\Rightarrow$ *Transfer* $\in$ events(t)
I6	EIP20	$\forall t \in Txs$: t=*approve* $\Rightarrow$ *Approval* $\in$ events(t)

Empirical Validation

- RQ1 How effective is SoCRATES in detecting invariant violations?
- RQ2 How does a society of bots compare to a single bot in detecting invariant violations?
- RQ3 What is the effectiveness of each atomic and combined bot behavior, in terms of number of invariant violations that bots can detect?
- RQ4 How does SoCRATES compare with Echidna, in terms of number of invariant violations?
- RQ5 How effective is SoCRATES in detecting contract-specific invariants?

Subject contracts

- Recent contracts
  - Last 10,000 transactions in the actual blockchain
- TOP contracts
  - Top 887 contracts by token capitalization, according to CoinMarketCap
- Filtering:
  - Open source contracts
  - Implementing EIP20 tokens
- Final set: 1,059 Recent + 846 TOP

Experimental settings

- `Ganace` default configuration
  - 10 accounts
  - Fresh block chain with no transaction
  - 10 repetitions, because of non-determinism
  - Timeout of 1,000 steps and 5 minutes

RQ1: Invariant violation detection

	I1		I2		I3		I4		I5		I6		TOTAL	
	FP	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**
Recent Top														
TOTAL														

RQ2: Society Vs single bot

	I1		I2		I3		I4		I5		I6		SUM	
Bots	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP
10														
1														

RQ3: Effect of bot behaviors

	I1		I2		I3		I4		I5		I6		TOTAL	
	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP
ALL														
Bound.+ Rand.														
Overflow														
Boundary														
Random														

Competitor tool(s)

- Echidna: fuzzing
  - User defined assertions (in Solidity)
- Manticore: symbolic execution
  - Limited support, overflow monitor unable to work on real contracts
- Oyente: symbolic execution
  - Old version of EVM, limited opcodes
- Mythril: static analysis
  - Does not emit test cases
- Madmax: static analysis
  - Does not emit test cases

RQ4: Comparison

- 10 contracts with implementation mistakes
- Manually edited to add invariant monitors
- Fuzzing with Echidna

Contract	Invariant	Result
ExpressCoin	I1	
AEToken	I1	
DNCEQuity	I1	
DELTAToken	I2	
Yumerium	I2	
Tube	I2	
JAAGCoin	I3	
MKC	I4	
CoinfairCoin	I5	
Bible	I6	

RQ5: Contract specific invariants

- New invariants that apply only to some contracts
- Randomly sample those that seem to implement additional features than EIP20
- The Oracle was integrated with these new Invariants

ID	Invariant
CSI1	$\forall t \in Txs$: owner → owner’ ⇒ owner=t.msg.sender
CSI2	$\forall t \in Txs$: totalSupply’ ≥ totalSupply
CSI3	$\sum_{a \in accounts}$ balanceOf(a) ≤ tokenLimit
CSI4	$\forall t \in Txs$: t=*enableTokenTransfer* ⇒ t.msg.sender=walletAddress
CSI5	totalAllocated ≤ (ADVISORS+FOUNDERS+HOLDERS+RESERVE)
CSI6	$\forall t \in Txs$: t=*getToken* ⇒ owner=t.msg.sender

Conclusion

A prominent case

- DAO: Decentralized Autonomous Organization
- Decentralized venture capital
- Receive Ether for DAO tokens
- $150M

- June 17, 2016 a loophole was found in the smart contact
- A hacker could steal $70M in DAO tokens
- Ethereum hard fork required

Multiple Roles

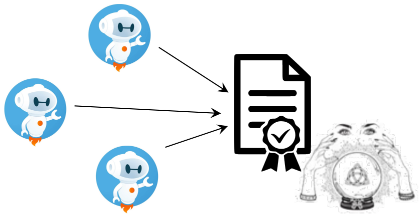
```
function buggedTransferFrom(
  address _from,
  address _to,
  uint256 _value
)
  public
  returns (bool)
{
  require(_value <= allowed[_from][msg.sender]);
  require(msg.sender != _from && _from != _to);

  balances[_from] -= _value;
  balances[_to] += _value;
  allowed[_from][msg.sender] -= _value;

  emit Transfer(_from, _to, _value);
  return true;
}
```

Novel Contribution

- Testing-based approach to identify implementation mistakes in smart contracts
  - Automated input generation to execute the contract
  - Automated oracle to detect invalid states at testing time
  - Society of bots to test a smart contract with different interactions



Empirical Validation

- RQ1 How effective is SoCRATES in detecting invariant violations?
- RQ2 How does a society of bots compare to a single bot in detecting invariant violations?
- RQ3 What is the effectiveness of each atomic and combined bot behavior, in terms of number of invariant violations that bots can detect?
- RQ4 How does SoCRATES compare with Echidna, in terms of number of invariant violations?
- RQ5 How effective is SoCRATES in detecting contract-specific invariants?