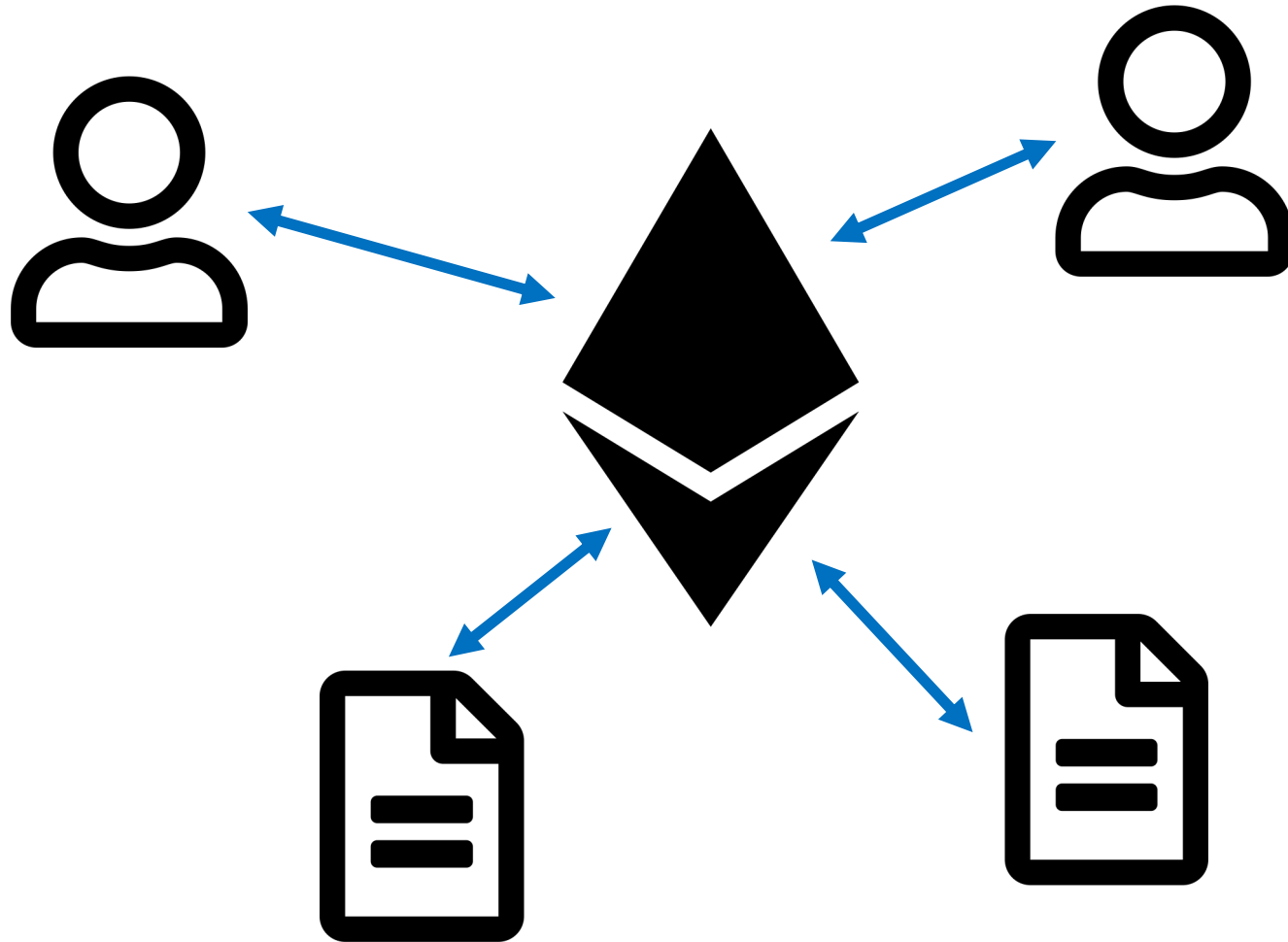


Detecting Standard Violation Errors in Smart Contracts

Fan Long

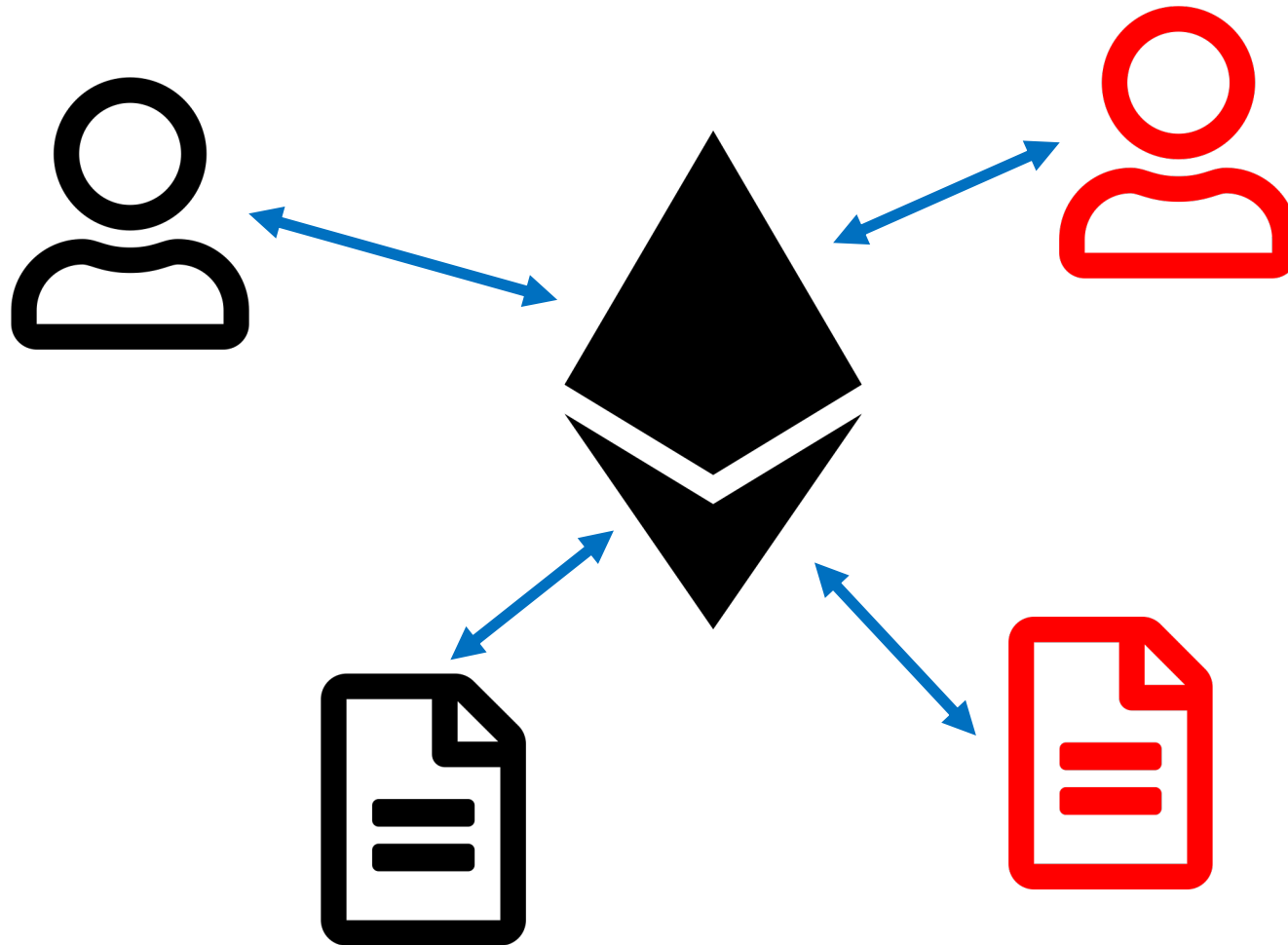
University of Toronto & Conflux Foundation

Smart Contracts

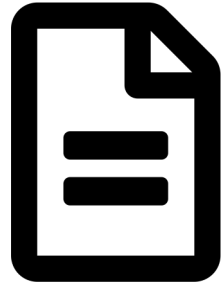


- Usages of Smart contracts
 - Tokens
 - Authorization
 - Poll
 - Lease agreement
 - ...

Ethereum and Smart Contracts



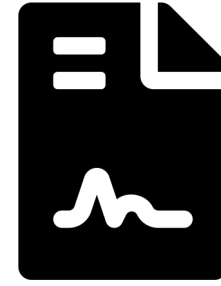
Smart Contracts



- Smart contracts
 - Tokens
 - Authorization
 - Poll
 - ...



Standards



- Standards
 - ERC-20, ERC-721
 - ERC-927
 - ERC-1417, ERC-1202
 - ...

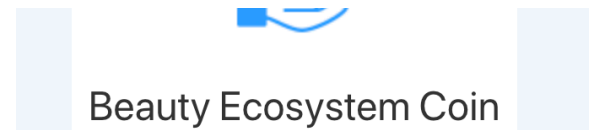


- Maker Token
- VeChain Token
- BECToken
- USD Coin
- ...

Standard Implementation

- Maker Token
- VeChain Token
- BECToken
- USD Coin
- ...

- Dai: The cryptocurrency with price stability that is the asset of exchange in the Dai Stablecoin System. It is a standard Ethereum token adhering to the ERC20 standard.



What can I do with UET?

UET is a standard ERC20 token, so you can hold it and transfer it.

Standard Implementation

- Maker Token
- VeChain Token
- BECToken
- USD Coin
- ...

Your Tokens Are Mine: A Suspicious
**OKEx exchange suspended BEC
withdrawal and trading because of
batchOverflow attack**

Multiple ERC20 Smart Contracts
(CVE-2018-11397, CVE-2018-11398)



What is BECToken?

- BECToken
 - A digital token claims that it satisfies ERC-20 standard.
 - Tokens can be transferred between addresses.
 - BECToken was attacked in April 2018. The market cap of BECToken evaporated in days.

ERC-20 Fungible Token

```
contract ERC20Interface {
```

```
    function totalSupply() public returns (uint);
```

```
    function balanceOf(address tokenOwner) public returns (uint);
```

```
    function transfer(address to, uint tokens) public returns (bool);
```

```
    function allowance(address tokenOwner, address spender) public returns
```

```
(uint);
```

- totalSupply(): the total supply of the token.

- balanceOf(): returns the balance of given account.

```
    function approve(address spender, uint tokens) public returns (bool success);
```

```
    function transferFrom(address from, address to, uint tokens) public returns
```

```
(bool);
```

```
}
```


ERC-20 Fungible Token

```
contract ERC20Interface {
```

```
function totalSupply() public returns (uint);
```

```
function balanceOf(address tokenOwner) public returns (uint);
```

```
function transfer(address to, uint tokens) public returns (bool);
```

```
function allowance(address tokenOwner, address spender) public returns  
(uint);
```

- transfer(): transfer the transaction sender's token to the receiver.

```
function approve(address spender, uint tokens) public returns (bool success);
```

```
function transferFrom(address from, address to, uint tokens) public returns  
(bool);
```

```
}
```

ERC-20 Fungible Token

```
contract ERC20Interface {
```

```
    function totalSupply() public returns (uint);
```

```
    function balanceOf(address tokenOwner) public returns (uint);
```

```
    function transfer(address to, uint tokens) public returns (bool);
```

```
    function allowance(address tokenOwner, address spender) public returns  
(uint);  
    function approve(address spender, uint tokens) public returns (bool success);  
    function transferFrom(address from, address to, uint tokens) public returns  
(bool);  
}
```

$$\sum_{a \in \text{Address}} (\text{balanceOf}(a)) = \text{totalSupply}()$$

What happened to their Implementation?

- BECToken

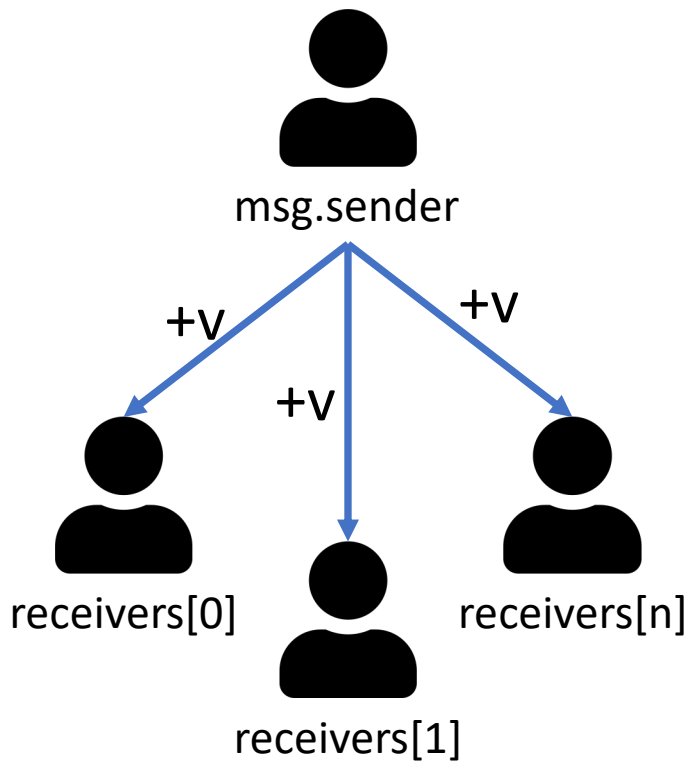
```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v;
    require(_value > 0 && balances[msg.sender] >= amount);
    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[receivers[i]] = balances[receivers[i]].add(v);
        Transfer(msg.sender, _receivers[i], v);
    }
}
```

What happened to their Implementation?

balances is a bookkeeping variable that tracks balances for each addresses.

```
mapping (address => uint256) balances;  
function batchTransfer(address[] receivers, uint256 v) public {  
    uint cnt = receivers.length;  
    uint256 amount = uint256(cnt) * v;  
    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(v);  
        Transfer(msg.sender, _receivers[i], v);  
    }  
}
```

What happened to their Implementation?



mapping (address => uint256) balances;

function batchTransfer(address[] receivers, uint256 v) **public** {

uint cnt = receivers.length;

uint256 amount = uint256(cnt) * v;

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(v);

Transfer(msg.sender, _receivers[i], v);

}

}

What happened to their Implementation?

The function first computes the total amount of token to be transferred.

```
mapping (address => uint256) balances;  
function batchTransfer(address[] receivers, uint256 v) public {  
    uint cnt = receivers.length;  
    uint256 amount = uint256(cnt) * v;  
    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(v);  
        Transfer(msg.sender, _receivers[i], v);  
    }  
}
```

What happened to their Implementation?

The function then updates the message senders balance.

```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v;
    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(v);
        Transfer(msg.sender, _receivers[i], v);
    }
}
```


What happened to their Implementation?

At last, the
function update
receivers' balances.

```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v;
    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(v);
        Transfer(msg.sender, _receivers[i], v);
    }
}
```


What happened to their Implementation?

$v=2^{255}$
receivers.length=2
amount = 0

```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v; 
    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(v);
        Transfer(msg.sender, _receivers[i], v);
    }
}
```


What happened to their Implementation?

$v=2^{255}$

receivers.length=2


amount = 0

```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v;
    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(v);
        Transfer(msg.sender, _receivers[i], v);
    }
}
```



What happened to their Implementation?

$v=2^{255}$
receivers.length=2
amount = 0

```
mapping (address => uint256) balances;
function batchTransfer(address[] receivers, uint256 v) public {
    uint cnt = receivers.length;
    uint256 amount = uint256(cnt) * v;
    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(v); 
        Transfer(msg.sender, _receivers[i], v);
    }
}
```

What happened to their Implementation?

- BECToken

```
uint256 amount = uint256(cnt) * v;  
require(_value > 0 && balances[msg.sender] >= amount);
```

The attacker could send a large amount of tokens that he or she does not own, effectively generating BECTokens from the air!

What happened to their Implementation?

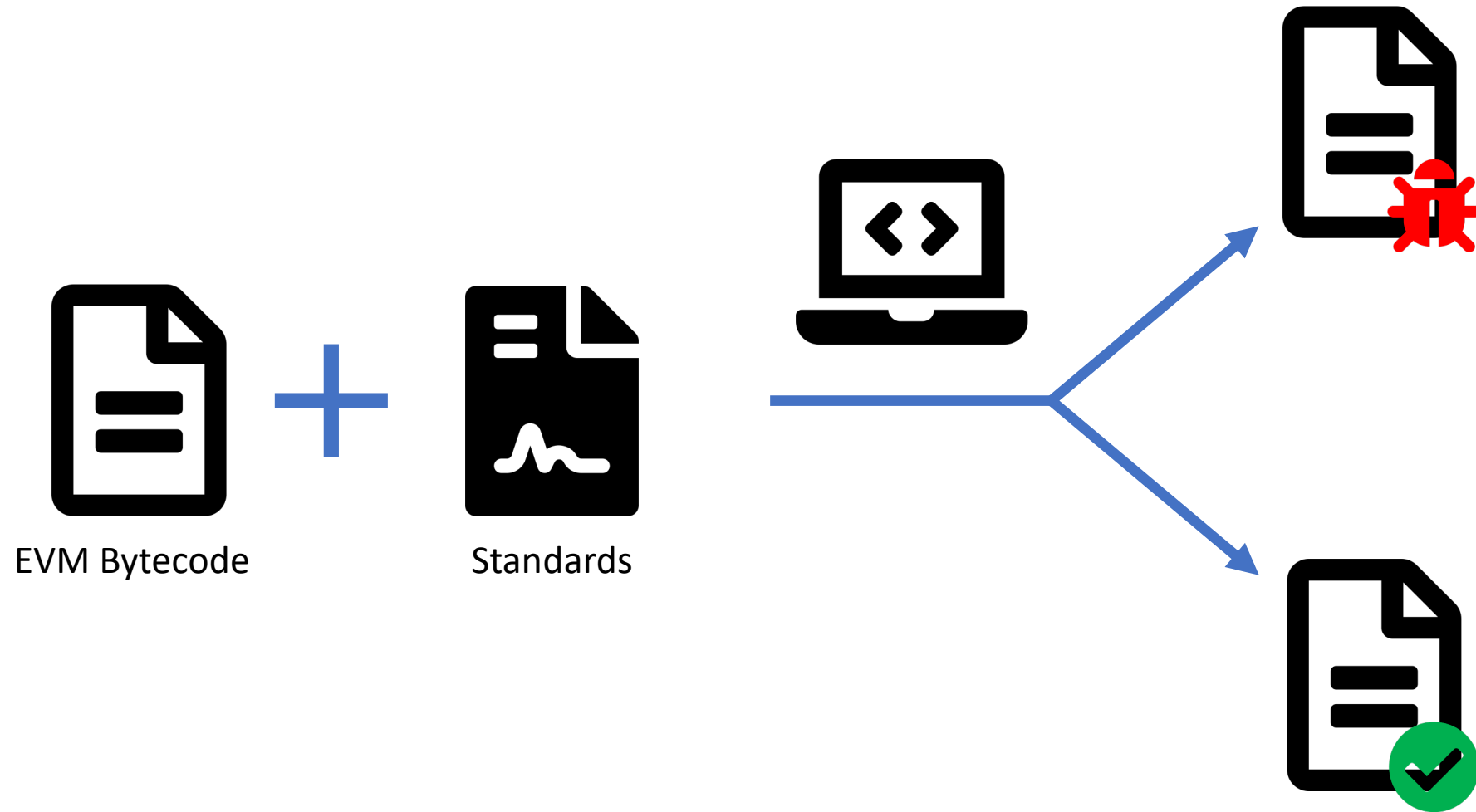
- BECToken

```
uint256 amount = uint256(cnt) * v;  
require(_value > 0 && balances[msg.sender] >= amount);
```

The sum of account balances equals to total supply!



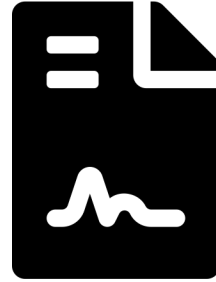
Solar



Total Supply Invariant

$$\sum_{a \in \text{Address}} (\text{balanceOf}(a)) = \text{totalSupply}()$$

Standard Specification



```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

```
    check(sum + bal >= sum)
```

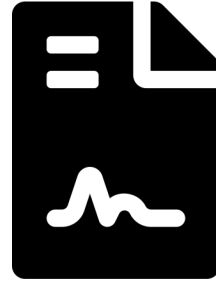
```
    sum += bal
```

```
check(sum == C.totalSupply())
```

- Solar allows user to specify constraints using a Python-like language.

$$\sum_{a \in Address} (balanceOf(a)) = totalSupply()$$

Standard Specification



```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

```
    check(sum + bal >= sum)
```

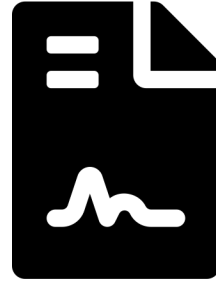
```
    sum += bal
```

```
check(sum == C.totalSupply())
```

- The function first computes the sum of account balances.

$$\sum_{a \in \text{Address}} (\text{balanceOf}(a)) = \text{totalSupply}()$$

Standard Specification



```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

```
    check(sum + bal >= sum)
```

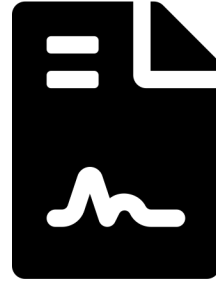
```
    sum += bal
```

```
check(sum == C.totalSupply())
```

- Helper variable ADDR represents the set of all possible addresses.

$$\sum_{a \in \text{Address}} (\text{balanceOf}(a)) = \text{totalSupply}()$$

Standard Specification



```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

```
    check(sum + bal >= sum)
```

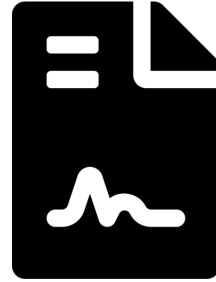
```
    sum += bal
```

```
check(sum == C.totalSupply())
```

- The function calls `balanceOf()` to retrieve the balance of each address.

$$\sum_{a \in \text{Address}} (\text{balanceOf}(a)) = \text{totalSupply}()$$

Standard Specification



```
sum = 0
for address in ADDRS:
    bal = C.balanceOf(address)
    check(sum + bal >= sum)
    sum += bal
check(sum == C.totalSupply())
```

- It then checks whether the sum of balances equals to the result returned by `totalSupply()`.

$$\sum_{a \in Address} (balanceOf(a)) = totalSupply()$$

Transfer Constraint

```
acc = [SymAddr(), SymAddr()]
assume(acc[0] != acc[1])
value = SymInt()
pre_bal = [c.balanceOf(account) for account in acc]
assume(pre_bal[0] + pre_bal[1] >= pre_bal[0])
result = c.transfer(acc[1], value, sender=acc[0])
post_bal = [c.balanceOf(account) for account in acc]
check(result == 0 and
      pre_bal[0] == post_bal[0] and
      pre_bal[1] == post_bal[1] or
      result != 0 and
      pre_bal[0] - value == post_bal[0] and
      pre_bal[1] + value == post_bal[1] and
      post_bal[0] >= pre_bal[0] and
      pre_bal[1] >= post_bal[1])
```

```
acc = [SymAddr(), SymAddr(), SymAddr()]
values = [SymInt(), SymInt()]
assume(acc[0] != acc[1] and acc[1] != acc[2])
```

- Transaction initiator has enough token.
- The balances of both sender and receiver are updated accordingly.

```
pre_bal = [c.balanceOf(account) for account in acc]
c.approve(acc[2], values[0], sender=acc[0])
assume(pre_bal[0] >= values[0])
r2 = c.transferFrom(acc[0], acc[1], values[1], sender=acc[2])
post_bal = [c.balanceOf(account) for account in acc]
check(pre_bal[0] - values[0] == post_bal[0] and
      pre_bal[1] + values[1] == post_bal[1] and
      post_bal[0] <= pre_bal[0] and
      post_bal[1] >= pre_bal[1] and
      values[0] >= values[1] and
      values[0] <= r3 or r2 == 0 and
      pre_bal[0] == post_bal[0] and
      pre_bal[1] == post_bal[1] and
      r3 == values[0])
```

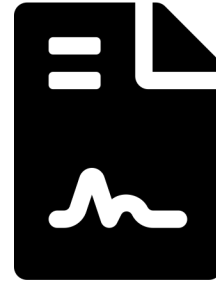
ERC-20 Fungible Token

```
contract ERC20Interface {
```

- Approve and transferFrom are two functions that allows the token owners to authorize a third party to spend their tokens.

```
function allowance(address tokenOwner, address spender) public returns  
(uint);  
function approve(address spender, uint tokens) public returns (bool success);  
function transferFrom(address from, address to, uint tokens) public returns  
(bool);  
}
```

Standard Specification



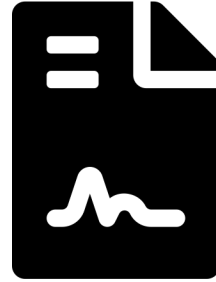
```
acc = [SymAddr(), SymAddr()]
```

- Transaction initiator has enough allowance.
- Token owner has enough balance.
- The balances of both sender and receiver are updated accordingly.

```
pre_bal[1] >= post_bal[1])
```

```
acc = [SymAddr(), SymAddr(), SymAddr()]
values = [SymInt(), SymInt()]
assume(acc[0] != acc[1] and acc[1] != acc[2])
pre_bal = [c.balanceOf(account) for account in acc[:2]]
assume(pre_bal[0] + pre_bal[1] >= pre_bal[0]) r1 =
c.approve(acc[2], values[0], sender=acc[0])
assume(r1 != 0)
r2 = c.transferFrom(acc[0], acc[1], values[1], sender=acc[2])
r3 = c.allowance(acc[0], acc[2])
post_bal = [c.balanceOf(account) for account in acc[:2]]
check(r2 != 0 and
pre_bal[0] - values[1] == post_bal[0] and
pre_bal[1] + values[1] == post_bal[1] and
r3 + values[1] == values[0] and
post_bal[0] <= pre_bal[0] and
post_bal[1] >= pre_bal[1] and
values[0] >= values[1] and
values[0] <= r3 or r2 == 0 and
pre_bal[0] == post_bal[0] and
pre_bal[1] == post_bal[1] and
r3 == values[0])
```

Standard Specification



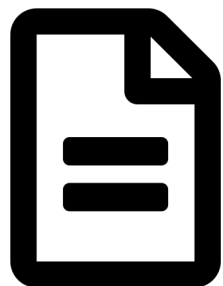
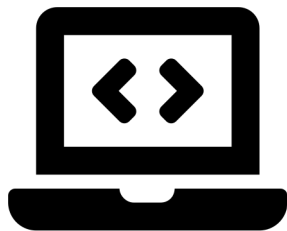
```
acc = [SymAddr(), SymAddr()]
assume(acc[0] != acc[1])
value = SymInt()
pre_bal = [c.balanceOf(account) for account in acc]
assume(pre_bal[0] + pre_bal[1] >= pre_bal[0])
```

- Transaction initiator has the allowance from the token owner.

```
post_bal = [c.balanceOf(account) for account in acc]
check(pre_bal[0] == post_bal[0] and
pre_bal[1] == post_bal[1] or
result != 0 and
pre_bal[0] - value == post_bal[0] and
pre_bal[1] + value == post_bal[1] and
post_bal[0] >= pre_bal[0] and
pre_bal[1] >= post_bal[1])
```

```
acc = [SymAddr(), SymAddr()]
tid = SymInt()
assume(acc[0] != acc[1])
owner = c.ownerOf(tid)
assume(owner != acc[1])
app = c.getApproved(tid)
is_approved = c.isApprovedForAll(owner, acc[0])
assume(a[0] != app) assume(is_approved == 0)
c.transferFrom(owner, a[1], tid, sender=acc[0])
pos_owner = c.ownerOf(tid)
check(pos_owner == acc[1])
```


Solar



EVM Bytecode



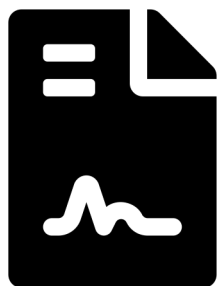
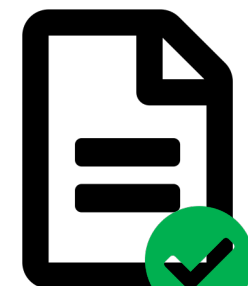
Transaction Stack



Symbolic Execution

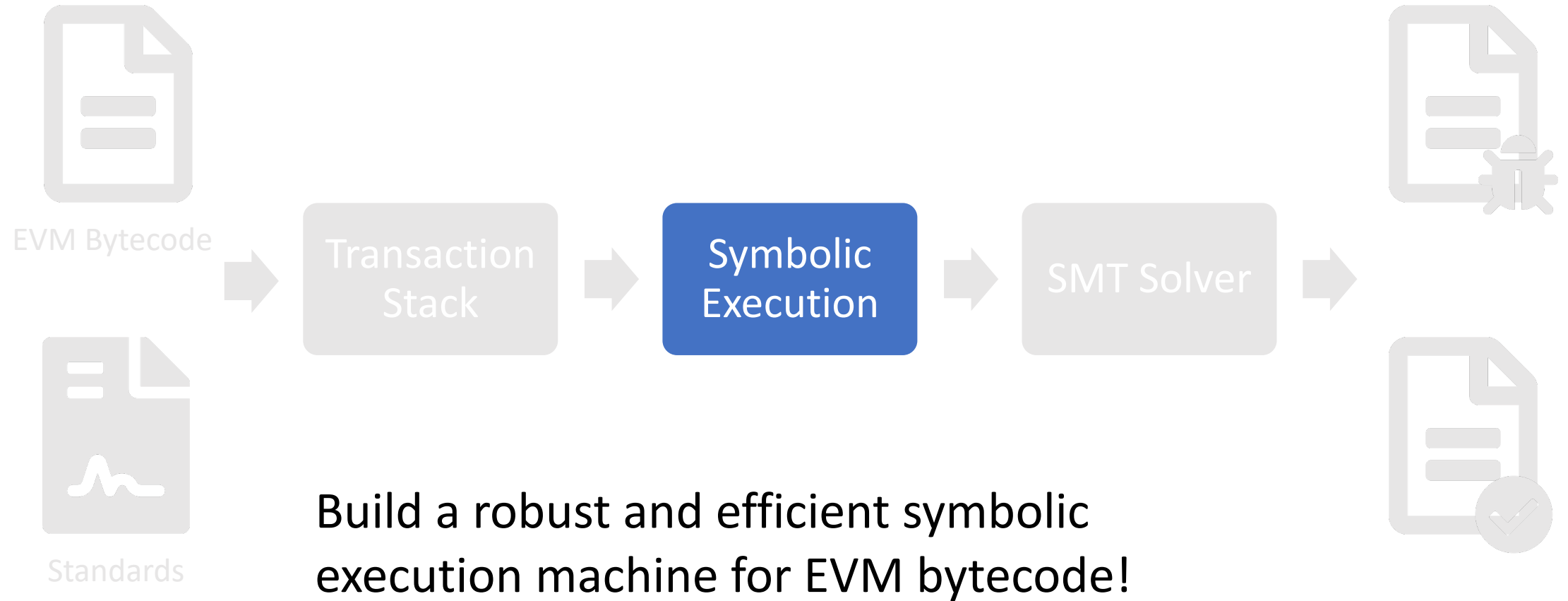


SMT Solver



Standards

Solar



Challenge: Address Scheme

- Solidity state address space:
 - 256bit address \rightarrow uint256
- Solidity uses crypto hash function to compute the storage location for dynamically allocated variables.
- Constraint solver cannot handle crypto computations efficiently.

Challenge: Address Scheme

```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

```
    check(sum + bal >= sum)
```

```
    sum += bal
```

```
check(sum == C.totalSupply())
```

```
uint256 totalSupply;  
mapping (address => uint256) balances;  
function balanceOf(address src) public view returns (uint) {  
    return balances[src];  
}
```

Storage Access Optimization

sum = 0

for address in ADDRS:

bal = C.balanceOf(address)

check(sum + bal >= sum)

sum += bal

check(sum == C.totalSupply())

```
uint256 totalSupply;
```

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

```
function balanceOf(address src) public view returns (uint) {
```

```
    return balances[src];
```

```
}
```

totalSupply



Persistent Storage

Challenge: Address Scheme

sum = 0

for address in ADDRS:

bal = C.balanceOf(address)

check(sum + bal >= sum)

sum += bal

check(sum == C.totalSupply())

```
uint256 totalSupply;  
mapping (address => uint256) balances;  
function balanceOf(address src) public view returns (uint) {  
    return balances[src];  
}
```

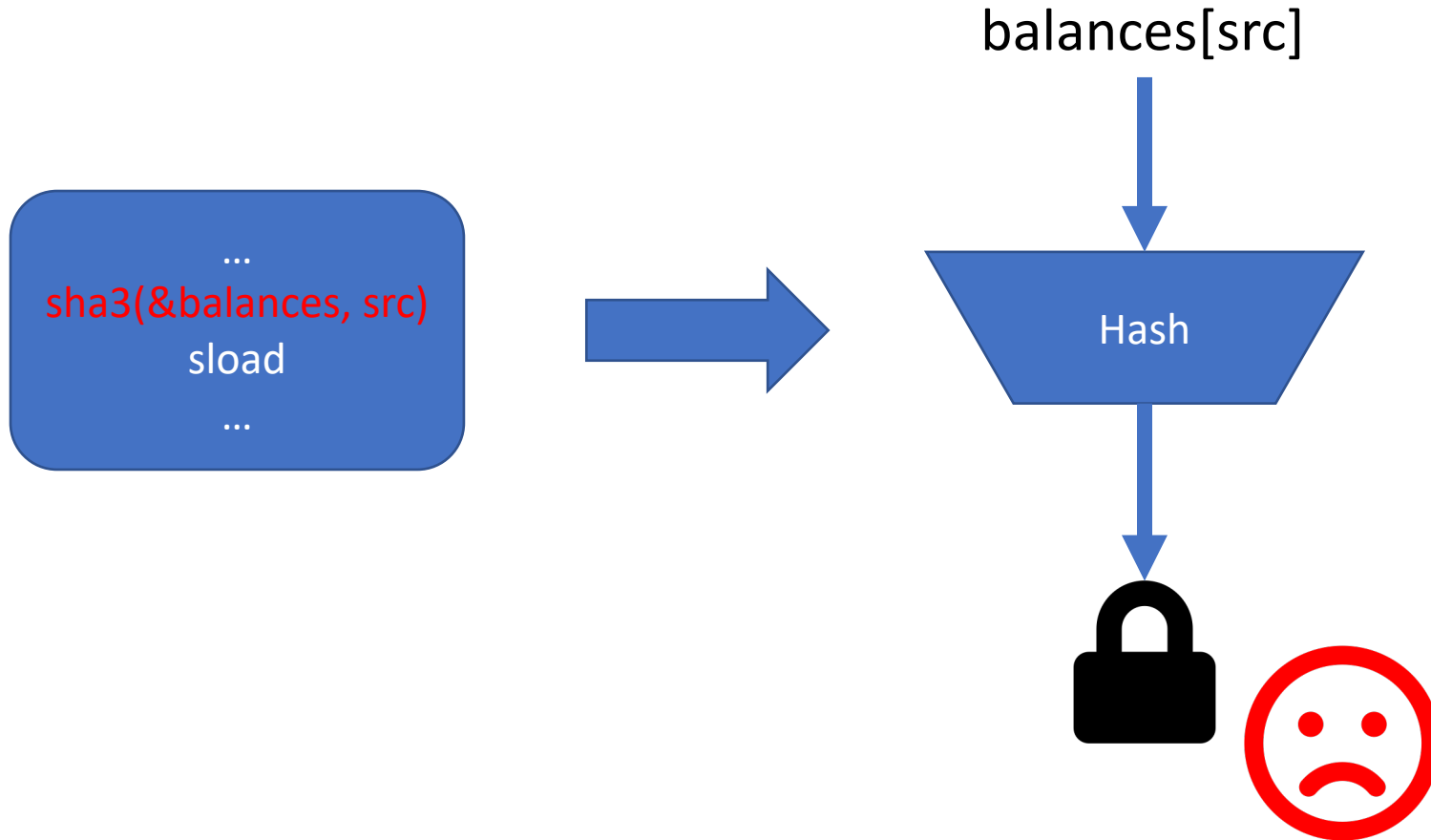
...
sha3(&balances, src)
sload
...

totalSupply



Persistent Storage

Storage Access Optimization

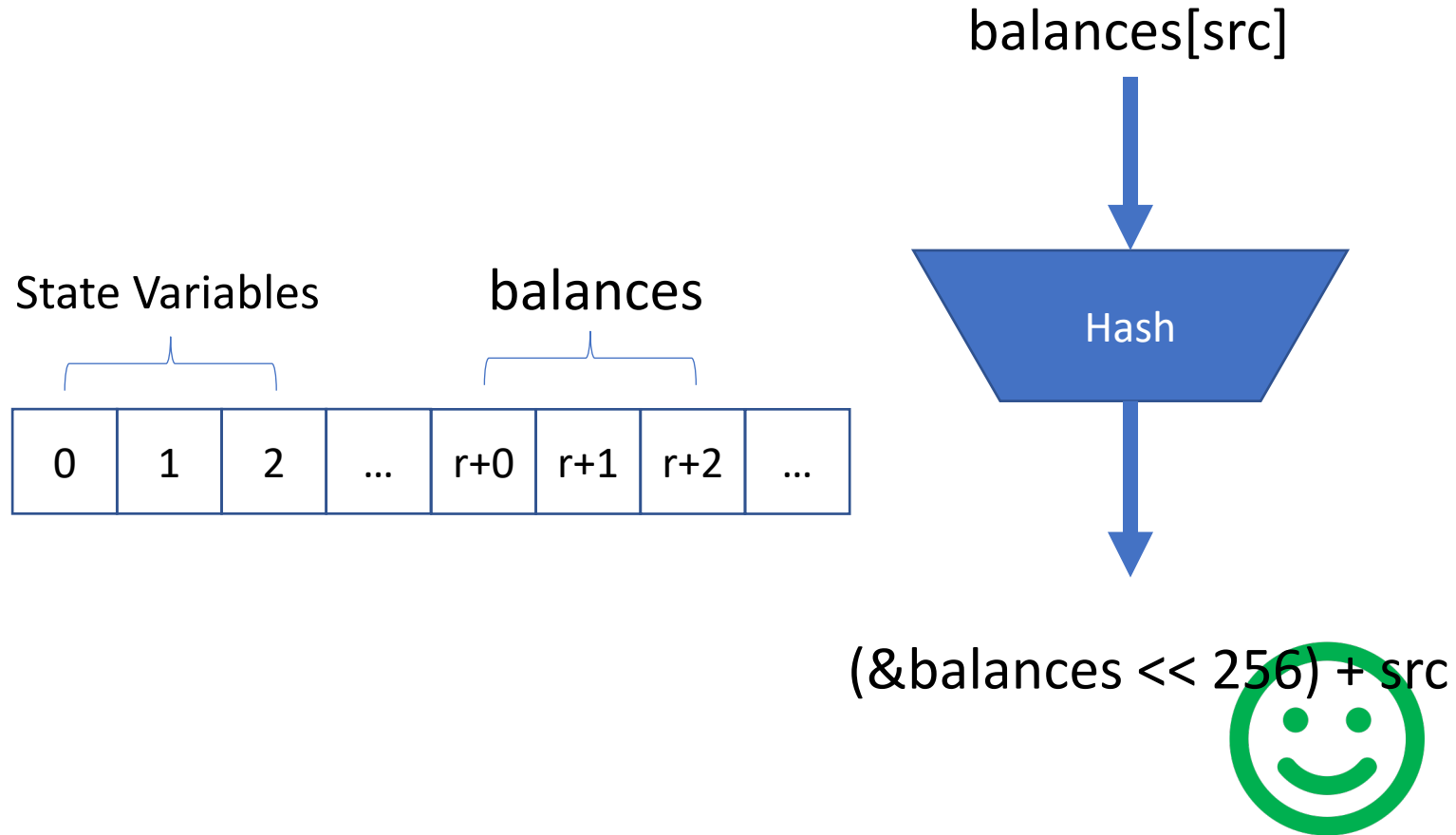


- Crypto hash function
- Avoid collision
- Expensive for solver

Our Solution

- Static analysis on the binary code to pair SHA3 with storage access operations.
- Change **every load/store** to use a customized address scheme that is equivalent to the original one (assuming no hash collision).
- Symbolic executes on the modified EVM byte code

Storage Access Optimization



- Customized address scheme
- Avoid collision
- Efficient for solver

Challenge: Volatile Memory

- Solidity state address space:
 - 256bit address → uint256
- Solidity volatile memory:
 - 256bit address → uint8
- Integers are broken into 32 bytes and then merged again when moving between state/volatile memory
- **Solution:** cache symbolic value stored into the volatile memory

Challenge: Account Addresses

```
sum = 0
```

```
for address in ADDRS:
```

```
    bal = C.balanceOf(address)
```

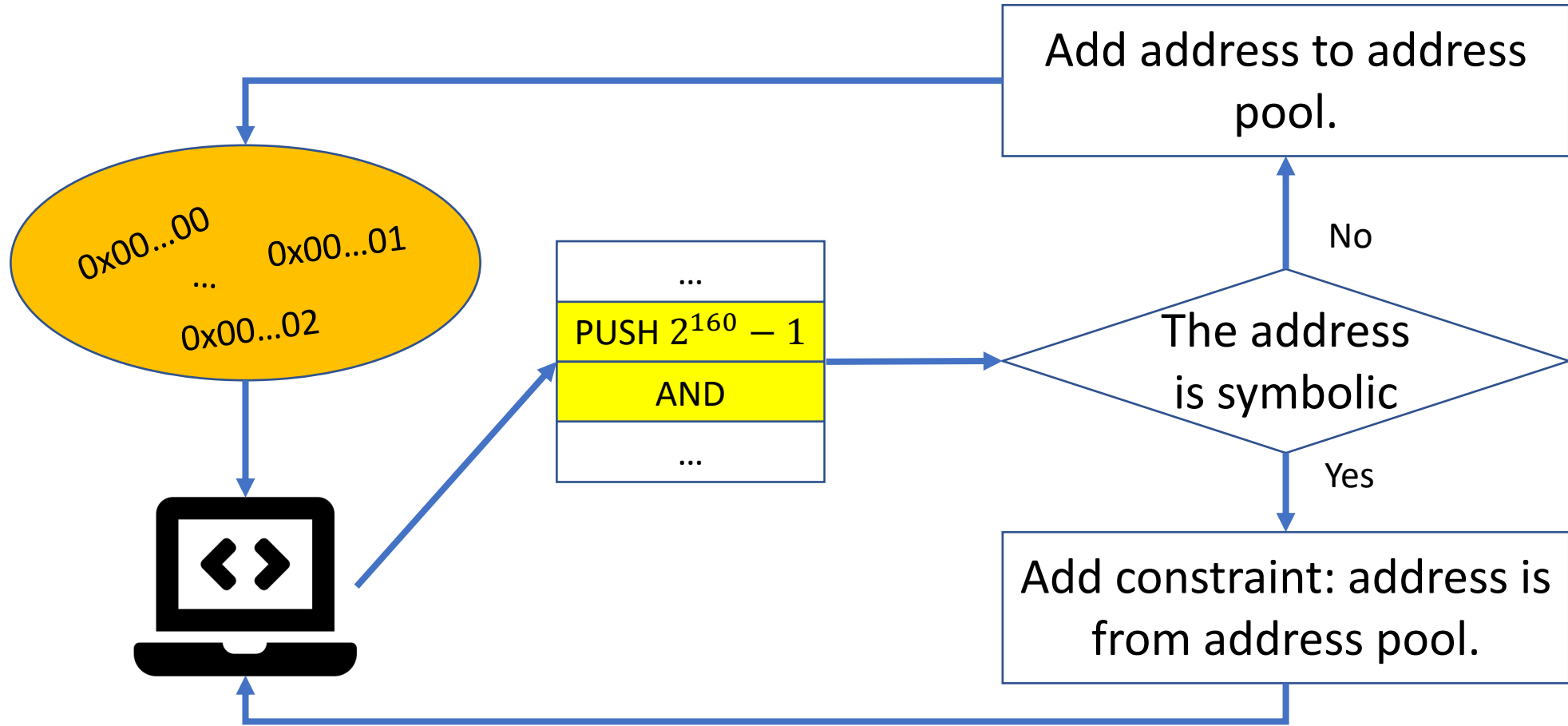
```
    check(sum + bal >= sum)
```

```
    sum += bal
```

```
check(sum == C.totalSupply())
```

- Address ranges from 0 to 2^{160}
- It is impossible to iterate over all possible addresses.

Account Address Pool



Evaluation

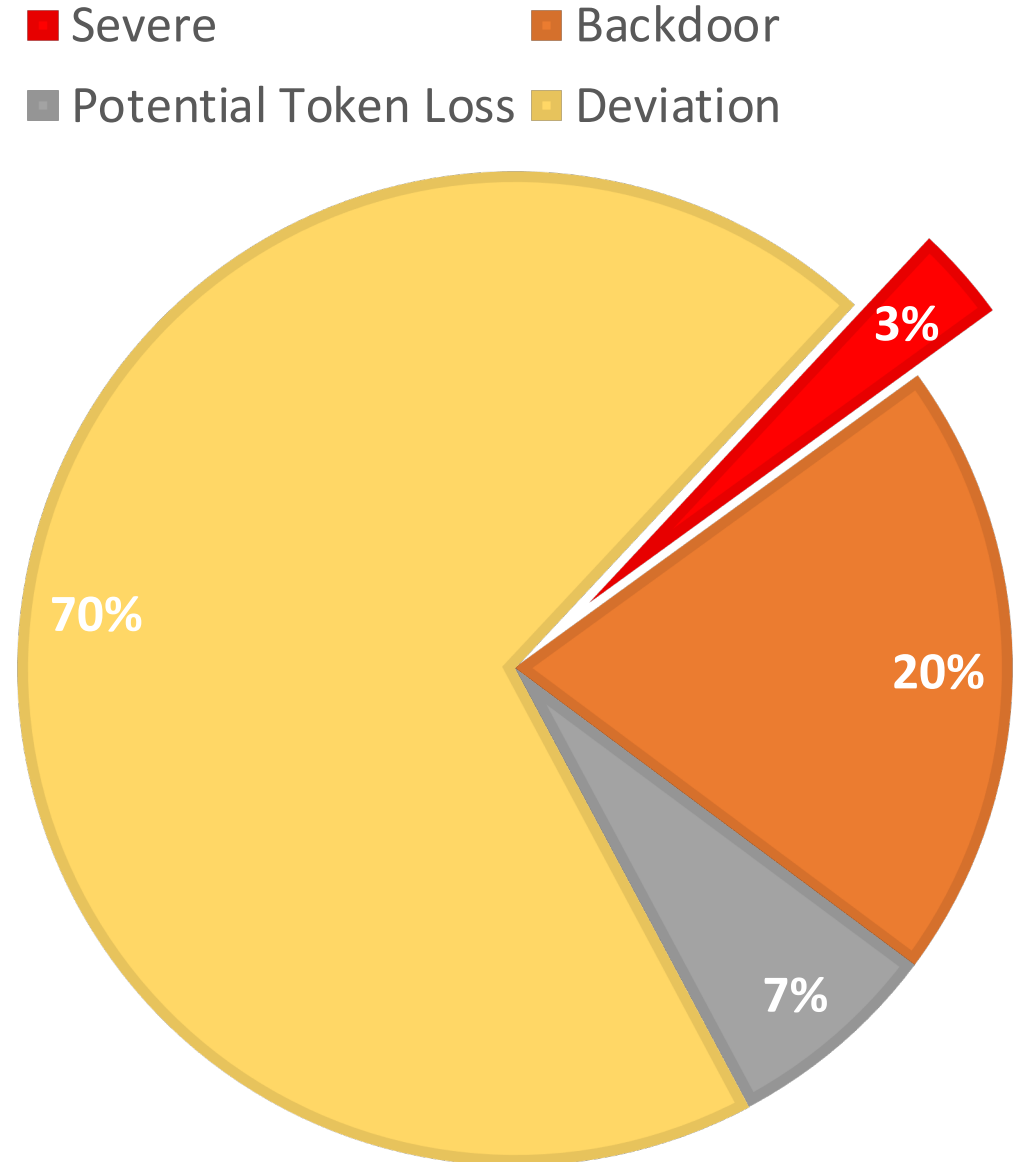
- 779 ERC-20 smart contracts from EtherScan
- 310 ERC-721 smart contracts from EtherScan
- Four Security Policies
 - ERC-20
 - Total Supply
 - Approve and TransferFrom
 - Transfer
 - ERC-721
 - Approve and TransferFrom

Evaluation

- 228 errors.
- 210 new errors.
- 188 vulnerable contracts.
- Only 10 false positives.

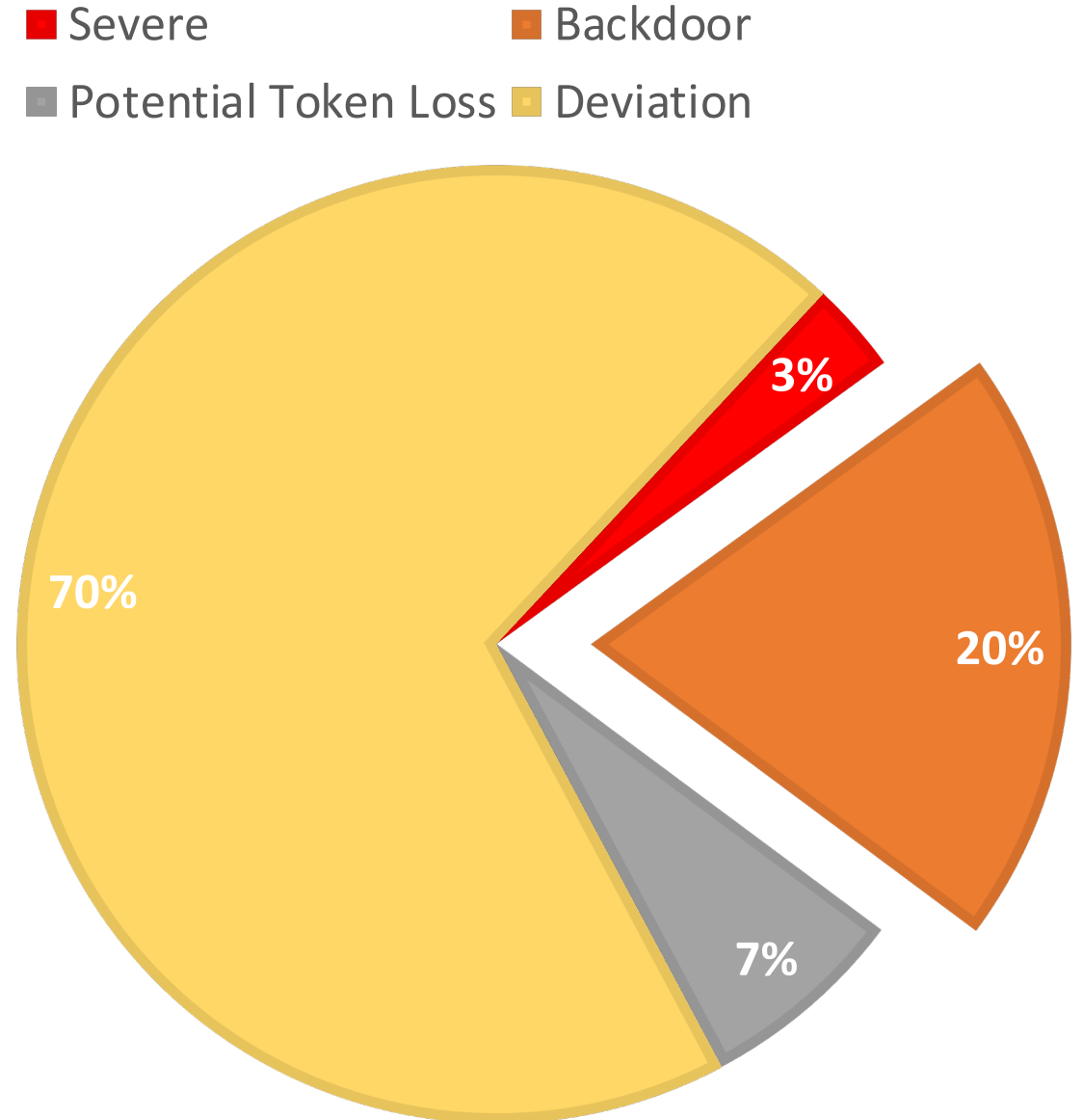
Evaluation

- Anyone
- Financial loss of contract participants



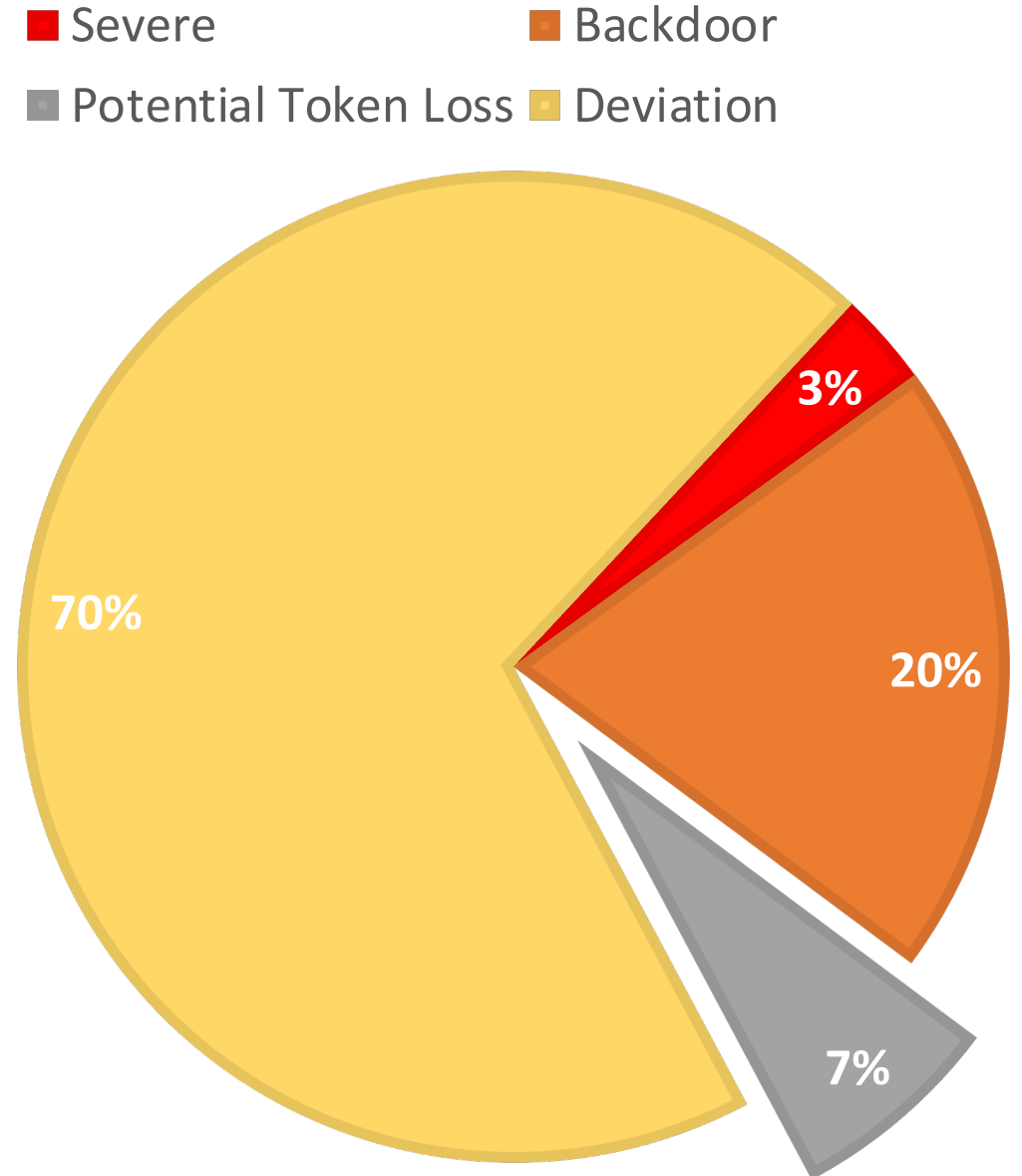
Evaluation

- Contract owner
- Exploitable privileges



Evaluation

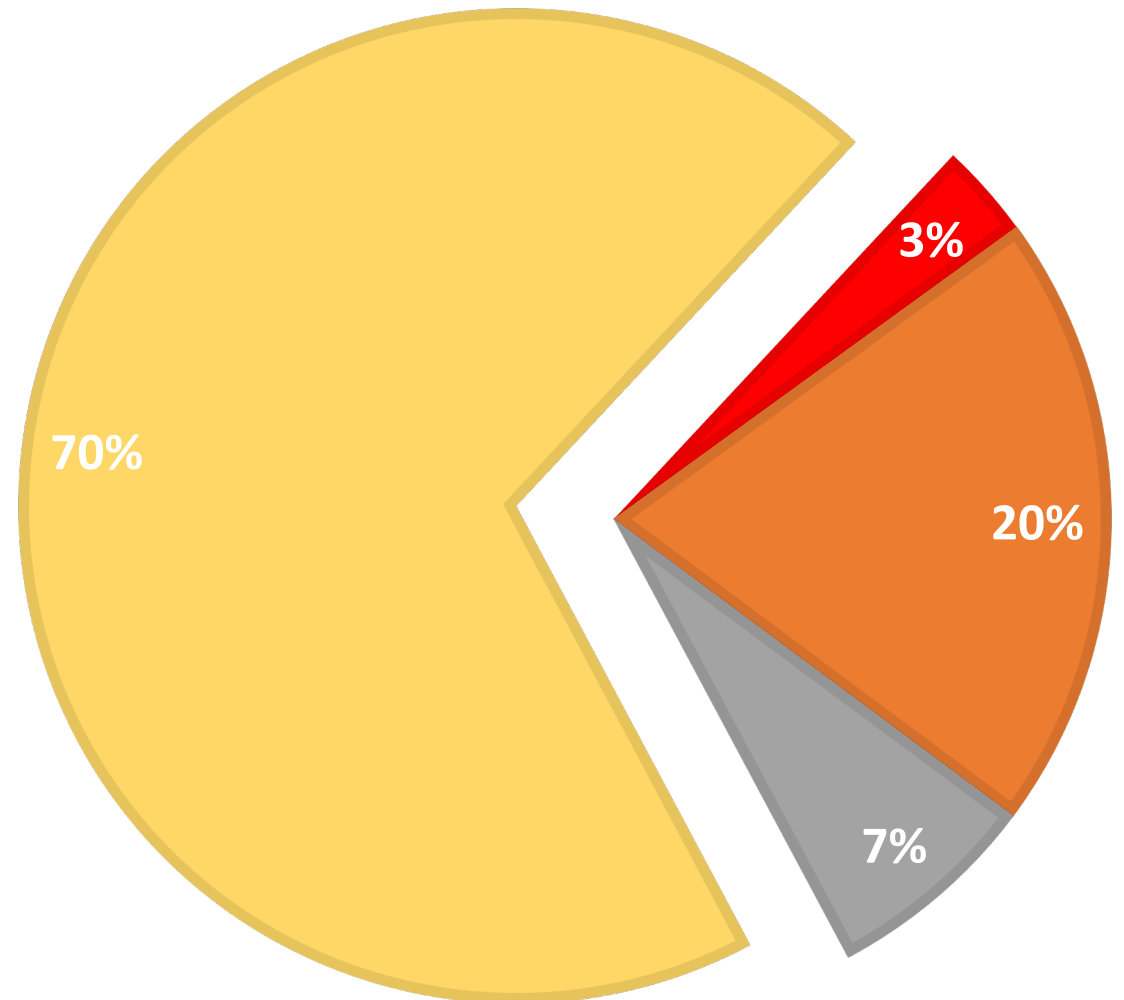
- Theoretically exploitable
- Specific time period
- A large amount of digital assets



Evaluation



- Extra functionalities



Comparison with Other Tools

- Sampled 100 smart contracts for manual analysis

	Whole Benchmark		100 Sampled Benchmark		
Tool	Reported Errors	Reported Contracts	True Positive	False Positive	Benign Errors
Securify	2432	518	1	183	85
Oyente	3036	763	7	198	85
Mythril	1627	730	3	63	61
Solar	228	188	25	2	0

- Solar reports **more true positives** and significantly **less false positives** and **benign errors**

Why Solar Performs Better?

- Utilizing standard information as specifications
 - Capable of detecting logic errors
 - No benign errors
- Optimized symbolic execution engine for EVM
 - Efficient and accurate handling of load/store instructions
 - Much less false positives

Example - Severe

```
function transferFrom(address from, address to, uint value) {  
...  
  if(value < allowance[to][msg.sender]) return false;  
...  Should be greater than or equal to (>)  
}
```

- This error allows an attacker to transfer one account's tokens to the other without proper approval.

Example - Backdoor

```
function mint(address _holder, uint _value) external {  
...  
    require(totalSupply + _value <= TOKEN_LIMIT);  
    balances[_holder] += _value;  
    totalSupply += _value;  
...  
}
```

Example - Backdoor

```
function mint(address _holder, uint _value) external {  
...  
    require(totalSupply + _value <= TOKEN_LIMIT);  
    balances[_holder] += _value;  
    totalSupply += _value;  
...  
}
```

- This error allows the contract owner to allocate more tokens than `TOKEN_LIMIT`.
- It also allows the contract owner to modify `_holders` balance to an arbitrary value.

Example – Potential Token Loss

```
function claimMigrate() {  
  balances[msg.sender] += pendingMigrations[msg.sender].amount;  
  ...  
}
```


Example – Potential Token Loss

```
function claimMigrate() {  
    balances[msg.sender] += pendingMigrations[msg.sender].amount;  
    ...  
}
```

- If the sender has large amount of token in previous contract, his/her balance will be overflowed.

Example – Deviation

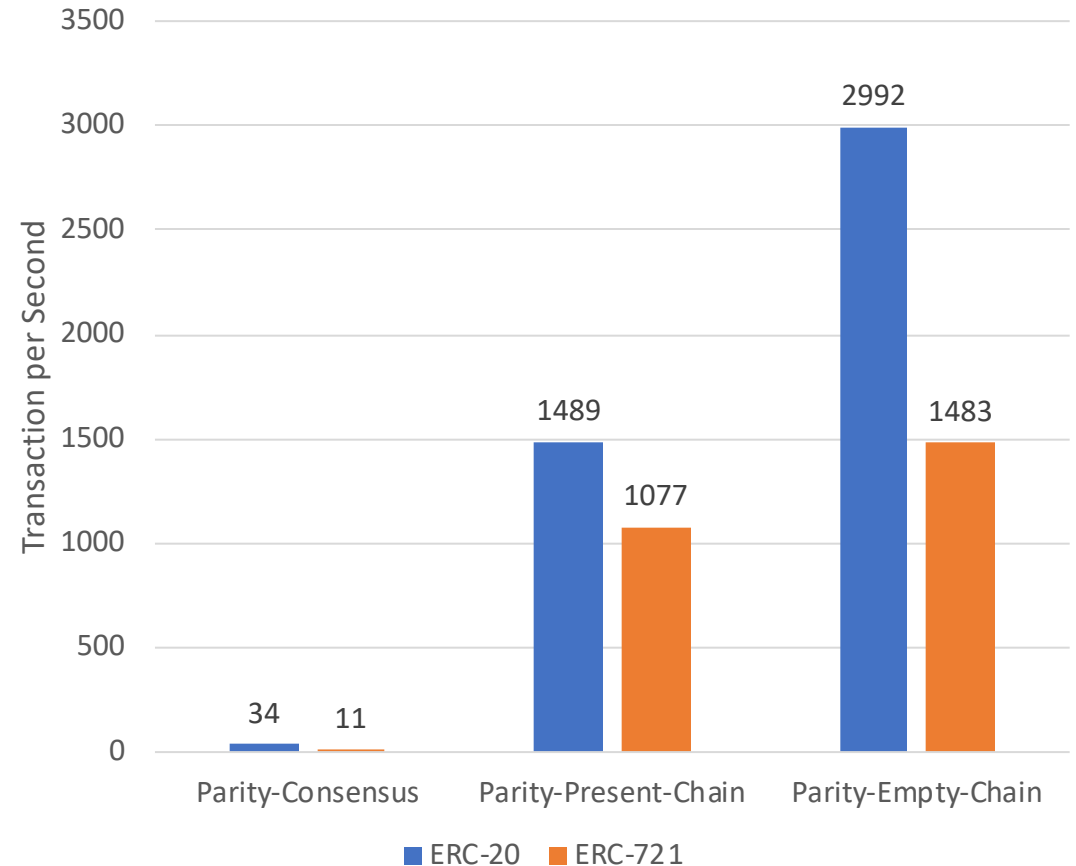
- Transfer function without return value.
 - Prevents other contract from calling transfer function.
- Frozen token.
 - Breaks the total supply invariant
- Standard deviation may lead to token loss depending on how the token is used.

Can we detect standard violation errors with no false positive and no false negative?

Yes! Runtime checks!

Consensus is the Primary Bottleneck

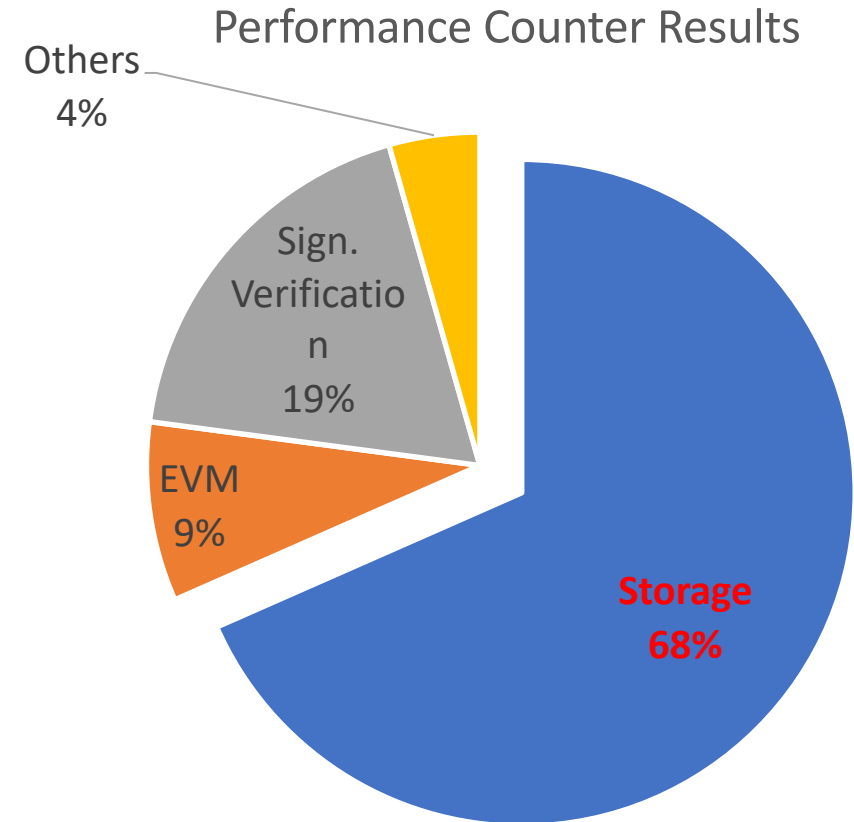
- Parity is one of the fastest Ethereum client
- Run ERC20/ERC721 transactions:
 - With normal Parity client
 - With Parity but without consensus
 - With Parity, without consensus, and with an empty blockchain state as the start
- Consensus limits the throughput with the block gas cap



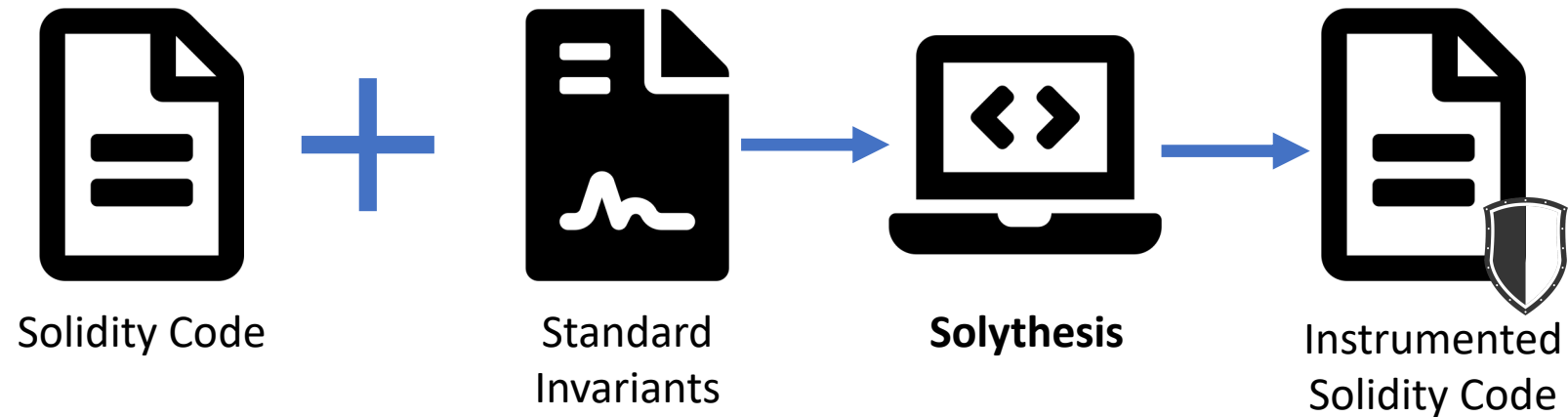
Running Parity with an empty chain is faster?

Storage is the Secondary Bottleneck

- Over 68% of performance counters are inside RocksDB or for load/store instructions
- Other EVM parts only take 9%
- Not all EVM instructions are equal
- State load/store instructions are significantly more expensive than other EVM instructions

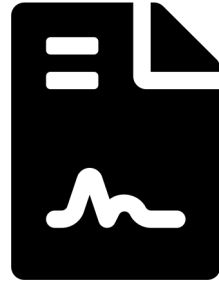


Solythesis

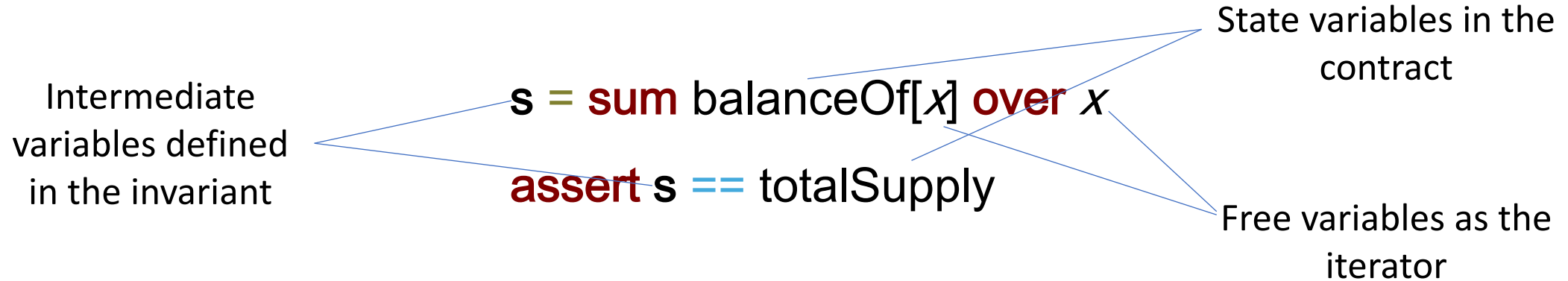


- Given standard invariants, Solythesis instruments Solidity code
- The instrumented code rejects transactions that violate invariants
- Design goal:
 - Minimize storage access instructions
 - Be expressive enough for all kinds of invariants

Solythesis Invariants



- ERC20 total supply invariants:



$$\sum_{a \in Address} (balanceOf(a)) = totalSupply()$$

ERC1202 Voting Contract Standard

- ERC1202 is a standard for smart contracts to implement voting
- It supports hosting **multiple issues**
- Each issue contains **multiple options** to vote
- Each participant may have a different **weight** for each issue
- For each issue, the option with the **highest accumulated weight** wins

- However, the example in ERC1202 contains an implementation error

ERC1202 Example

```
mapping (uint => mapping (address => uint256)) weights;  
mapping (uint => mapping (uint => uint256)) weightedVoteCounts;  
mapping (uint => mapping (address => uint)) ballot;  
function vote(uint issued, uint option) public {  
    uint256 weight = weights[issued][msg.sender];  
  
    weightedVoteCounts[issued][option] += weight;  
  
    ballots[issued][msg.sender] = option;  
}
```

ERC1202 Example

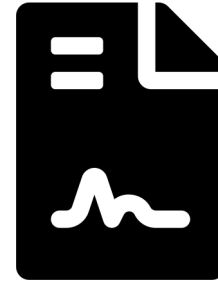
```
mapping (uint => mapping (address => uint256)) weights;  
mapping (uint => mapping (uint => uint256)) weightedVoteCounts;  
mapping (uint => mapping (address => uint)) ballot;  
function vote(uint issued, uint option) public {  
    uint256 weight = weights[issued][msg.sender];  
    weightedVoteCounts[issued][option] += weight;  
  
    ballots[issued][msg.sender] = option;  
}
```

Problem: People may
vote multiple times!

ERC1202 Example

```
mapping (uint => mapping (address => uint256)) weights;  
mapping (uint => mapping (uint => uint256)) weightedVoteCounts;  
mapping (uint => mapping (address => uint)) ballots;  
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
    weightedVoteCounts[issueId][ballots[issueId][msg.sender]] -= weight;  
    weightedVoteCounts[issueId][option] += weight;  
  
    ballots[issueId][msg.sender] = option;  
}
```

ERC1202 Solythesis Invariant



- The weightedVoteCounts should always equal to the sum of the weights of participants who voted for the option

s = **map** *a,b* **sum** weights[*a*][*x*] **over** *x* **where** ballot[*a*][*b*] == *x*
forall *a,b* **assert** **s**[*a*][*b*] == weightedVoteCounts[*a*][*b*]

- **s** is an intermediate map that conditionally sums over expressions
- The combination of **assert** and **forall** defines constraints that iterate over all elements of maps

How to Efficiently Enforce Such Invariant?

- **Naïve Approach:** Loops over all relevant map values in the blockchain state to check the invariant at the end of every transaction
 - Extremely slow
 - High gas cost
- **Our Approach:** Synthesize **delta updates** to intermediate values and **delta invariant check** to evaluate relevant constraints
 - Instrument runtime checks only for values that might change!

Delta Update

$s = \text{map } a, b \text{ sum weights}[a][x] \text{ over } x \text{ where ballot}[a][b] == x$

- Declare a new map (uint -> uint -> uint) to maintain the value of s .
- Synthesize and instrument code to update s when:
 - weights is updated
 - or ballot is updated

Delta Update

```
function vote(uint issueId, uint option) public {  
  uint256 weight = weights[issueId][msg.sender];  
  weightedVoteCounts[issueId][option] += weight;  
  
  ballots[issueId][msg.sender] = option;  
  
}
```



Solythesis computes the binding between quantifier variables and contract expression:
 $a \rightarrow \text{issueId}$
 $b \rightarrow \text{msg.sender}$
 $x \rightarrow \text{ballot}[\text{issueId}][\text{msg.sender}]$

$s = \text{map } a, b \text{ sum } \text{weights}[a][x] \text{ over } x \text{ where } \text{ballot}[a][b] == x$

Delta Update

```
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
    weightedVoteCounts[issueId][option] += weight;  
    s[issueId][ballot[issueId][msg.sender]] -= weights[issueId][msg.sender];  
    ballots[issueId][msg.sender] = option;  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
}
```


Delta Invariant Check

forall a, b **assert** $s[a][b] == \text{weightedVoteCounts}[a][b]$

- Only check relevant instances of (a,b) when:
 - s is updated
 - or $\text{weightedVoteCounts}$ is updated
- Maintain lists to track relevant instances

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
  
    weightedVoteCounts[issueId][option] += weight;  
  
    s[issueId][ballot[issueId][msg.sender]] -= weights[issueId][msg.sender];  
    ballots[issueId][msg.sender] = option;  
  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
}
```

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
    a_arr.push(issueId); b_arr.push(option);  
    weightedVoteCounts[issueId][option] += weight;  
  
    s[issueId][ballot[issueId][msg.sender]] -= weights[issueId][msg.sender];  
    ballots[issueId][msg.sender] = option;  
  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
}
```

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
    a_arr.push(issueId); b_arr.push(option);  
    weightedVoteCounts[issueId][option] += weight;  
    a_arr.push(issueId); b_arr.push(ballot[issueId][msg.sender]);  
    s[issueId][ballot[issueId][msg.sender]] -= weights[issueId][msg.sender];  
    ballots[issueId][msg.sender] = option;  
  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
}
```

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    uint256 weight = weights[issueId][msg.sender];  
    a_arr.push(issueId); b_arr.push(option);  
    weightedVoteCounts[issueId][option] += weight;  
    a_arr.push(issueId); b_arr.push(ballot[issueId][msg.sender]);  
    s[issueId][ballot[issueId][msg.sender]] -= weights[issueId][msg.sender];  
    ballots[issueId][msg.sender] = option;  
    a_arr.push(issueId); b_arr.push(ballot[issueId][msg.sender]);  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
}
```

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    ...  
    a_arr.push(issueId); b_arr.push(ballot[issueId][msg.sender]);  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
  
}
```

Delta Invariant Check

```
function vote(uint issueId, uint option) public {  
    ...  
    a_arr.push(issueId); b_arr.push(ballot[issueId][msg.sender]);  
    s[issueId][ballot[issueId][msg.sender]] += weights[issueId][msg.sender];  
    for (uint256 index = 0; index < a_arr.length; index +=1)  
        assert (s[a_arr[index]][b_arr[index]] ==  
                weightedVoteCounts[x_arr[index]][y_arr[index]]);  
}
```

More Optimizations

- **Volatile Memory:** Volatile memory is much cheaper than state load/store. We replace states with volatile memory whenever possible.
- **Cache Load:** If a state variable is loaded multiple times, we will remove future loads and cache it in the volatile memory
- **Eliminate Redundant Updates:** Eliminate those instrumentations that are redundant

Solythesis Experiments

- We collect three representative contracts:
 - ERC20: BEC Token
 - ERC721: DozerDoll
 - ERC1202: Vote Example
- Apply Solythesis to instrument these contracts
- Run these contracts on Parity and measure the overhead
 - For BEC and DozerDoll, we use history transactions in Ethereum
 - For Vote Example, we synthesize a transaction trace that repeatedly call important functions like CreateIssues() and Vote()

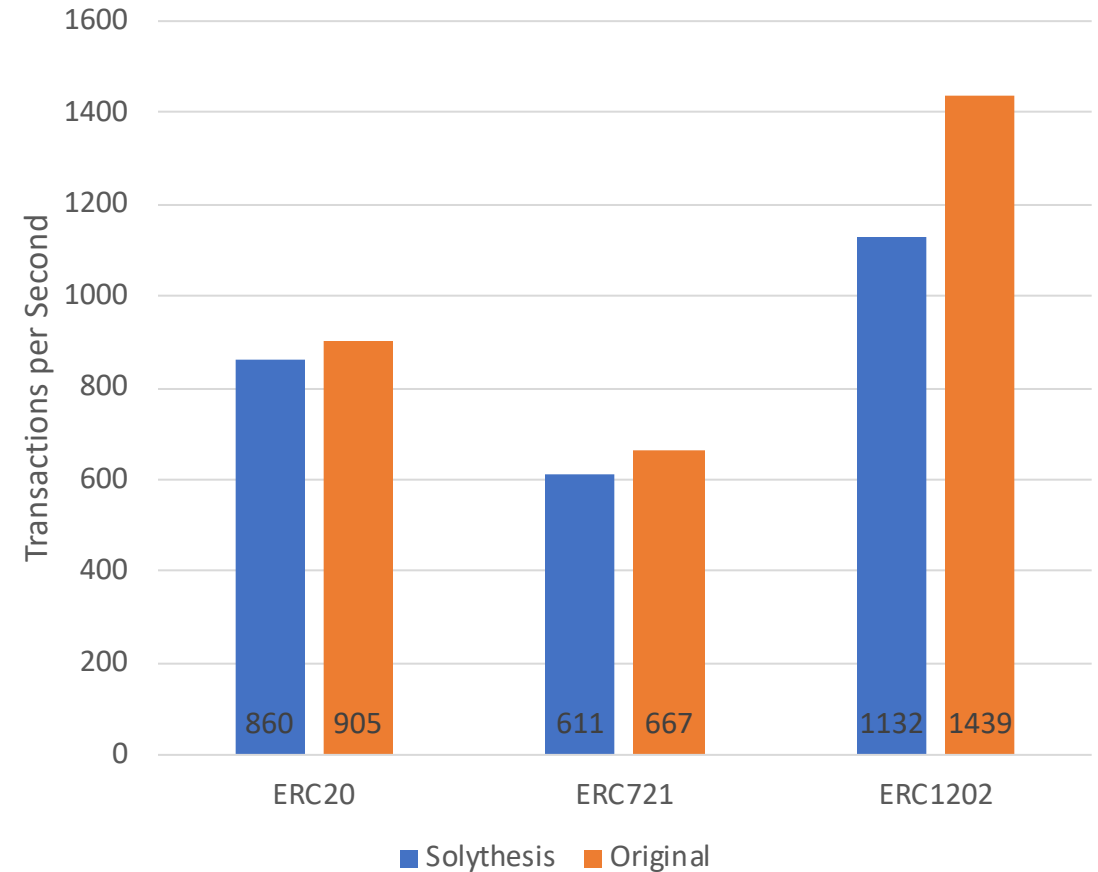
Results with Ethereum Consensus

		ERC-20	ERC-721	ERC-1202
Average CPU	Solythesis	1.534%	1.661%	2.810%
	Original	1.446%	1.681%	2.508%
Disk Write	Solythesis	42K/s	58K/s	82K/s
	Original	42K/s	54K/s	70K/s

- Comparing to expensive cost of running PoW consensus
- Negligible CPU usage increasement
- Negligible extra disk writes
- ~30% more gas for the instrumentation

Results without Ethereum Consensus

- Less than 5% overhead for ERC20
 - ~8% overhead for ERC721
 - ~20% overhead for ERC1202
-
- The overhead is tied to the number of instrumented loads/stores.



Conclusion

- Two tools for utilizing specifications from contract standards
 - Solar: Symbolic execution engine for EVM with significantly less false positive
 - Solythesis: Efficient runtime check instrumentation for Solidity code
- EVM is often the enemy for designing efficient program analysis.
 - SHA3 for addressing the space
 - Different layouts between the state space and the volatile memory space
- Smart contract execution environment is totally different from general purpose programs.
 - Consensus and storage are the bottleneck.
 - Different tradeoffs between performance and security