Certifying functional correctness of Ethereum smart contracts

Dr. Petar Tsankov
Co-founder and Chief scientist, ChainSecurity
Senior researcher, ICE center, ETH Zurich
@ptsankov
Inter-disciplinary research center at the #1 CS department in Europe

Next-generation blockchain security using automated reasoning

https://chainsecurity.com
@chain_security
What do these have in common?

Must not fail!

```solidity
contract Token {
    mapping(address => uint) balances;
    function balanceOf(address a) {
        return balances[a];
    }
    function transfer(address to, uint n) {
        balances[msg.sender] -= n;
        balances[to] += n;
    }
}
```
What sets **them apart**?

Certified using formal verification

```solidity
contract Token {
    mapping(address => uint) balances;

    function balanceOf(address a) {
        return balances[a];
    }

    function transfer(address to, uint n) {
        balances[msg.sender] -= n;
        balances[to] += n;
    }
}
```

Best-effort
What sets them apart?

Certified using formal verification

Token {
    mapping (addr => uint) balances;
    function balanceOf (address a) {
        return balances[a];
    }
    function transfer (address to, uint n) {
        balances[msg.sender] -= n;
        balances[to] += n;
    }
}

Best-effort

What sets them apart?

anyone can kill your contract #6

devops199 opened this issue a day ago · 12 comments

devops199 commented a day ago · edited

I accidentally killed it.
https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4
Our mission

Bring formal security guarantees to contracts

- Mathematically model **all** behaviors of smart contracts
- **Prove** that no bugs can occur
- Scale via **automation** and state-of-the-art research
Our mission

Bring formal security guarantees to contracts

- Formal verifier for certifying custom functional specifications of Ethereum contracts
- Prove specifications of Ethereum contracts
- Scale via automation and state-of-the-art research
Why is it hard to certify the custom behavior of smart contracts?

Note:
Find generic vulnerabilities ≠ Certify custom behavior
## Functional correctness

### Crowdsale

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint raised;</code></td>
<td>Raised funds</td>
</tr>
<tr>
<td><code>uint goal;</code></td>
<td>Goal</td>
</tr>
<tr>
<td><code>uint closeTime;</code></td>
<td>Close time</td>
</tr>
</tbody>
</table>

```solidity
function invest() { .. }
function close() { .. }
```

### Escrow

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapping(address =&gt; uint) deposits;</code></td>
<td>Deposits mapping</td>
</tr>
</tbody>
</table>

```solidity
function deposit() { .. }
function withdraw() { .. }
function claimRefund() { .. }
```

## Requirements

- Sum of all deposits equals the escrow’s ether balance
- Investors cannot claim refunds after the goal is reached
Step 1: **Formalize** requirements

(Informal) requirement:

"Sum of all deposits equals the escrow’s ether balance"

Formal property

always \( \text{sum(Escrow.deposits)} == \text{Escrow.balance} \)
Step 2: **Check** formal property
Methods and guarantees

Manual review

- Time consuming
- Can miss errors
Methods and guarantees

- Automated testing
  - Fuzzing
  - Symbolic execution
  - Can miss errors

- Manual review
  - Time consuming
  - Can miss errors
**Fuzzing**

**Tools:** ChainFuzz, Echidna, ContractFuzzer, Harvey, ...
Symbolic execution

**Tools:** Oyente, Manticore, Mythril, MAIAN, ...
Methods and guarantees

Formal verification
- Automated program verification
- Proves absence of errors

Automated testing
- Fuzzing
- Symbolic execution
- Can miss errors

Manual review
- Time consuming
- Can miss errors
Formal verification

VerX
Automated formal verification with VerX

“Investors can claim refunds only if the sum of deposits never exceeded 10,000 ether “

Smart contract

```solidity
mapping(address => uint) deposits;
function claimRefund(){..}
```

Formal property

```solidity
(always Escrow.claimRefund
==> !before(sum(deposits) >= 10000)
```
Expressive and intuitive specifications

Access control

always Escrow.deposit(address)

==> (msg.sender == Escrow.owner)
# Expressive and intuitive specifications

| Access control | always $\text{Escrow.deposit(address)}$  
|                | $\implies (\text{msg.sender} == \text{Escrow.owner})$ |
| State-based properties | always $(\text{now} > \text{Vault.refundTime} + 1 \text{ week})$  
|                      | $\implies ! \text{Vault.refund(uint256)}$ |
### Expressive and intuitive specifications

| Access control                  | always Escrow.deposit(address)  
|                                |  
|                                |   ==> (msg.sender == Escrow.owner)   
| State-based properties         | always (now > Vault.refundTime + 1 week)  
|                                |  
|                                |   ==> ! Vault.refund(uint256)   
| State machine properties      | always !(once(state == REFUND)  
|                                |  
|                                |   && once(state == FINALIZED)   

**State machine properties**

- always !(once(state == REFUND)
- && once(state == FINALIZED)
## Expressive and intuitive specifications

| Access control | always Escrow.deposit(address)  
|                | ==> (msg.sender == Escrow.owner) |
| State-based properties | always (now > Vault.refundTime + 1 week)  
|                | ==> ! Vault.refund(uint256) |
| State machine properties | always !(once(state == REFUND)  
|                | && once(state == FINALIZED)) |
| Invariants over aggregates | always totalSupply == sum(balances) |
## Expressive and intuitive specifications

<table>
<thead>
<tr>
<th>Category</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access control</strong></td>
<td><code>always Escrow.deposit(address) == (msg.sender == Escrow.owner)</code></td>
</tr>
<tr>
<td><strong>State-based properties</strong></td>
<td><code>always (now &gt; Vault.refundTime + 1 week) == ! Vault.refund(uint256)</code></td>
</tr>
<tr>
<td><strong>State machine properties</strong></td>
<td><code>always !(once(state == REFUND) &amp;&amp; once(state == FINALIZED))</code></td>
</tr>
<tr>
<td><strong>Invariants over aggregates</strong></td>
<td><code>always totalSupply == sum(balances)</code></td>
</tr>
<tr>
<td><strong>Multi-contract invariants</strong></td>
<td><code>always Token.totalSupply &gt;= Sale.issuance</code></td>
</tr>
</tbody>
</table>
Expressive and intuitive specifications

| Access control | always Escrow.deposit(address)  
|                | => (msg.sender == Escrow.owner) |
| State-based properties | always (now > Vault.refundTime + 1 week)  
|                      | => ! Vault.refund(uint256) |
| State machine properties | always (! once(state == REFUND))  
|                        | && once(state == FINALIZED)) |
| Invariant aggregates | always totalSupply == sum(balances) |
| Multi-contract invariants | always Token.totalSupply >= Sale.issuance |
Dealing with *unbounded* state spaces

Use *symbolic* (not concrete) values

Use *program abstraction*

Initial state

```
invest(X)  claimRefund(Y)
```

Bounded depth

Feasible width
Sound symbolic reasoning

- Hash-based storage allocation
- Gas mechanics
- Calls to untrusted contracts
- Dynamically constructed contracts
Impact and experience

**Fast** and **scalable** formal verification of Ethereum contracts
(157+ contracts, 100+ properties, ~1 min / property)

Benefits:
- **Certify** what works (go beyond bug finding)
- **Re-use** libraries of common specifications
- **Re-certification** is cheap
How to get access to VerX?

Demo:  http://verx.ch

VerX as a service:  contact@chainsecurity.com
One more announcement...
First automated framework for testing Solidity compilers
First automated framework for testing Solidity compilers

Internal compiler error for call to unimplemented "super" function #5130

nweller opened this issue on 2 Oct 2018 · 10 comments

nweller commented on 2 Oct 2018 · edited

The following is the output from the last recent build:
contract contractname { fun funcname { } contractname funfuncname { } contractname funfuncname { } }

It does compile and is recognized as an error in the Etherscan tool.

Exponentiation producing inconsistent results #4893

nweller opened this issue on 4 Sep 2018 · 6 comments

nweller commented on 4 Sep 2018 · edited

Across multiple unsigned integer types I’ve tried – e.g. uint8 – the exponentiation operator can produce a result which is outside of the range of that type in some contexts, but exhibits the truncation behavior I’d expect in other contexts.

I get different failure modes in the truffle-based test framework I’m using (with ganache-cli, Linux, solc-js 0.4.24 – this produces unexpected values) and http://remix.ethereum.org/ (this produces a VM error).

See:
First automated framework for testing Solidity compilers

https://github.com/eth-sri/soltix

https://discord.gg/XKSVavS
Investors can claim refunds only if the sum of deposits never exceeded 10,000 ether.

Smart contract mapping (
address => uint) deposits;
function claimRefund(){..}
(always Escrow.claimRefund
==> !before(sum(deposits) >= 10000)

Formal property Verified
May not hold

Methods and techniques

RELIABILITY
- Formal verification
  - Automated program verification
    - Proves absence of errors
- Automated testing
  - Fuzzing
  - Symbolic execution
    - Can miss errors
- Manual review
  - Time consuming
    - Can miss errors

Symbolic reasoning + abstraction

Bounded depth
Feasible width

Initial state
invest(X) claimRefund(Y)

invest(X)

claimRefund(Y)

VerX: Automated formal verification

VerX: Automated formal verification

"Investors can claim refunds only if the sum of deposits never exceeded 10,000 ether."