Deductive Verification of Smart Contracts

Mooly Sagiv
Deductive Verification of Smart Contracts

A Tribute to the Legacy of Prof. Zohar Manna 1939-2018
Zohar Manna
Weizmann Inst. & Stanford Uni.
1939-2018
Milestones

• Thesis (CMU, 1968)
• Mathematical Theory of Computation (1974)
• Logical Basis for Computer Programming (with Waldinger, 1985-8)
• STeP: A tool for deductive verification of systems
• Consummate teacher and advisor
Software verification

• The programmer defines what is the desired behavior
• Ensures there is a proof of correctness
• Proof covers all scenarios
Why verify smart contracts?
Smart Contracts are hard to get right

A barrier to trust!
Correctness is essential for Smart Contracts

**Traditional software**
- Buggy code is a reality
- Mechanisms for reverting effects of erroneous code execution
- Continuous code maintenance is standard practice

**Smart Contracts on Blockchains**
- Code as law
- Transactions are irreversible, often anonymous
- *Smart Contracts are unpatchable*
  - Upgrade is tricky
Auditing

- The standard procedure for checking contracts
- Expensive $$$
- Quality depends on the auditors
- Miss bugs
- Not decentralized
  - Auditor reputation is the trust authority
Semi-automatic deductive verification

Verification tool

System

Argument

Desired behavior

Incorrect Finds bug

Not sure

Correct Finds proof

UNDECIDABILITY
Find bugs or prove their absence
• No false alarms or missed errors

Define what is required from contracts
• Generic properties
  • No overflow
  • Isolation between contracts [POPL’18]

• Standard requirements
  • ERC20, ERC721
    Money market, Exchanges...

• Contract-specific correctness
  • Wallet should have sufficient number of signers
  • Correct libraries

[POPL’18] S. Grossman et. al. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts
“Compound worked with Certora to verify the correctness of a preliminary-version of a core contract.

The tool demonstrated a unique capability to discover not just the obvious corner-cases, but also subtle cases that would have been difficult, if not impossible, to find through standard unit-testing.

The Certora team discovered two subtle bugs in the contract which were patched, as well definitively proving a conjecture which influenced an important design decision.

Certora’s collection of properties proven to hold for all inputs and environments greatly increased our confidence in the correctness of our contract.”
Certora - Automatic Exact Verification (AEV)

Benefits

Superior Accuracy
Most accurate method to detect bugs

Automatic
No customization per contract or services are required

Zero False Alarms
All reported errors are real and come with risk explanation

Zero Missed Errors
All errors are eventually detected and come with formal checkable proofs
How does Certora-AEV work?

Smart Contracts → Compiler → EVM/eWASM → CERTORA AEV

Verification Report
Test cases to show bugs
Hoare Triples

- Useful to explain verification
- Annotate the code with assertions
- \(\{P\}\) Contract \(\{Q\}\)
  - Every execution of the contract starting in a state in \(P\) results in a state in \(Q\)

if \(P\) then \{
  Contract;
  assert \(Q\);
\}
Hoare Triples

- Useful to explain verification
- Annotate the code with assertions
- \{P\} Contract \{Q\}
  - Every execution of the contract starting in a state in \(P\) results in a state in \(Q\)

```plaintext
if P then {
  Contract;
  assert Q;
}
```
**Example Hoare Triples**

- `{ x=0 } x := x+2 { x=2 }  ✓ valid
- `{ x=0 } x := x+2 { x>0 }  ✓ valid
- `{ x=1 } x := x+y { x>0 }  ✗ invalid  Test y = -3
- `{ true } if x<0 then y:=-x else y:=x { y≥0 }  ✓ valid
- `{ y≥0 } t:=y; z:=1; while t>0 do z:=z*x; t:=t-1; done { z=x^y }  ✓ valid
Composing Operations

• Prove that
  \{P\}
  \text{command}_1 ;
  \text{command}_2 
  \{Q\}

• Find an intermediate assertion \( R \) and show
  • \{P\} \text{command}_1 \{R\}
  • \{R\} \text{command}_2 \{Q\}?

• Can be found automatically
Composing Operations Example

- How to prove that \( \{ x=0 \} \ x:=5 \{ ? \} \ y:=x+1 \{ y>0 \} \)
- Prove that
  - \( \{ x=0 \} \ x:=5 \{ x\geq0 \} \)
  - \( \{ x\geq0 \} \ y:=x+1 \{ y>0 \} \)
Example Wallet

```java
{ count(m_own) = 0 }

wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
```
wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        i++;

assert count(m_own) = own.len
```python
{ count(m_own) = 0 }

wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
```

Flowchart:
- Initial state: `m_own = {}`
- `own = [a, a]`
- Loop condition: `i < own.len`
- If `i < own.len` is true, then:
  - `m_own[own[i]] = true`
  - `i := i + 1`
- If `i < own.len` is false, then:
  - Assert `count(m_own) = own.len`
```plaintext
{ count(m_own) = 0 }  

wallet_constructor(address[] own)  
    int i = 0;  
    while (i < own.len)  
        m_own[own[i]] = true;  
        ++i;  

{ count(m_own) = own.len }  
```

Diagram:
- `own=[a,a]`  
- `m_own={}`  
- `i := 0`  
- `i < own.len`  
- `m_own[own[i]] = true`  
- `i := i+1`  
- `assert count(m_own)=own.len`
{ count(m_own) = 0 }

wallet_constructor(address[] own)
  int i = 0;
  while (i < own.len)
    m_own[own[i]] = true;
    ++i;

{ count(m_own) = own.len }

assert count(m_own)=own.len
```latex}
\{ \text{count}(m_{\text{own}}) = 0 \} \\

\text{wallet\_constructor}(\text{address[\[] \text{ own}}) \\
\text{int } i = 0; \\
\text{while } (i < \text{own}.\text{len}) \\
\quad m_{\text{own}}[\text{own}[i]] = \text{true}; \\
\quad ++i; \\

\{ \text{count}(m_{\text{own}}) = \text{own}.\text{len} \} \\
```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

assert count(m_own) = own.len
```python
wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

assert count(m_own) = own.len
```
{ count(m_own) = 0 }

wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }

assert count(m_own)=own.len
Fixed Wallet

{ count(m_own) = 0 }

wallet_constructor_fixed(address[] own)
    int i = 0;
    while (i < own.len)
        if (m_own[own[i]])
            abort;
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
Fixed Wallet

{ count(m_own) = 0 }

wallet_constructor_fixed(address[] own)
    int i = 0;
    while (i < own.len)
        if (m_own[own[i]])
            abort;
        m_own[own[i]] = true;
        ++i;

{ count(m_own) = own.len }
{ count(m_own) = 0 }

int i = 0;
while (i < own.len) { own.len \geq i \geq 0 \land count(m\_own) = i }
    if (m\_own[own[i]])
        abort;
    { own.len \geq i \geq 0 \land count(m\_own) = i \land \neg m\_own[own[i]] } 
    m\_own[own[i]] = true;
    { own.len \geq i \geq 0 \land count(m\_own) = i+1 }
    ++i;
    { own.len \geq i \geq 0 \land count(m\_own) = i }
{ count(m\_own) \geq own\_len }
How does Certora-AEV automatically check correctness?
Secret Sauce – Compilation and Constraint Solving

Smart Contracts

Front End

Verification Condition

Constraint Solver

Solution: Bug

No Solution: Proof
Secret Sauce – Compilation and Constraint Solving

Front End

\[(z > 0 \land x = 1) \lor (z \leq 0 \land x = 1)\] 
\[(t > z \land y = 1) \lor (z \leq 0 \land y = 1)\]
\[x \neq y\]

Constraint Solver

\[\text{ensures } x = y\]
Summary

• Ensured correctness is critical for the adoption of Smart Contracts
• Formal verification is the tool we have
• Enabling technologies
  • Modularity
  • Mature tools