Deductive Verification of Smart Contracts

Mooly Sagiv
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
Automatic deductive verification

Verification tool

- Incorrect: Finds bug
- Not sure
- Correct: Finds proof

System

Desired behavior

UNDECIDABILITY
What We Do: Technology for certifying contracts

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.”
int X = 100
dec(a){
    ... log(a) ...
}
inc(b){
    ... X += b ...
}

int Z = 4
log(b){
    ... Z = b ...
}

int T=0
f(q,e) {
    ... f(t, e) ...
}

f(1,2)
dec(100)
int X = 100
dec(a) {
    ... log(a) ...
}
inc(b) {
    ... X += b ...
}

int Z = 4
log(b) {
    ... Z = b ...
}

int T=0
f(q,e) {
    ... f(t, e) ...
}
int X = 100
dec(a){
    ... log(a) ...
}
inc(b){
    ... X += b ...
}

log(b){
    ...
    dec(b) ...
    ...
}

int T=0
f(q,e) {
    ... f(t, e) ...
}
Encapsulation ➔ Isolation?

```plaintext
module Blue

int X := 100

void dec(a)
    requires X >= 0
    if (X >= a)
        log(a)
        X := X - a
    ensures X >= 0
```
module Blue

int X := 100

void dec(a)
    requires X >= 0
    if (X >= a)
        log(a)
        X := X - a
    ensures X >= 0
module Blue

int X := 100

void dec(a)

requires X >= 0
if (X >= a)

log(a)

X := X - a

ensures X >= 0
module Blue

int X := 100  \hspace{1cm} X \geq 0

void dec(a)
  requires X >= 0
  if (X >= a)
    \log(a)  \hspace{1cm} X \geq 0 \land X \geq a
    X := X - a
  ensures X >= 0
module Blue

int X := 100

void dec(a)

  requires X >= 0
  if (X >= a)
    log(a)
    X := X - a
  ensures X >= 0

X >= 0
X >= 0 ∧ X >= a
module **Blue**

```plaintext
int X := 100

void dec(a)
    requires X >= 0
    if (X >= a)
        log(a)
        X := X - a
    ensures X >= 0
```

module **Red**

```plaintext
void log(a)
    if (*)
        dec(a)
```
module **Blue**

```plaintext
int X := 100

void dec(a)
    if (X >= a)
        log(a)
        l: X := X - a
```

module **Red**

```plaintext
void log(a)
    if ( * )
        dec(a)
```

**Store:**

<table>
<thead>
<tr>
<th>Blue</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=100</td>
<td>X=0</td>
</tr>
</tbody>
</table>

**Stack/Trace:**

<table>
<thead>
<tr>
<th>Blue</th>
<th>Red</th>
<th>Blue</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec(100)</td>
<td>log(100)</td>
<td>dec(100)</td>
<td>log(100)</td>
</tr>
</tbody>
</table>
Effective Callback Freedom (ECF)

- Correctness condition for modularity
- Tames callbacks – do not add behaviors
- Enforces isolation between contracts
- Not syntactic
- Does not preclude good behaviors
A Non ECF Execution for Blue

module Blue
    int X := 100
    void dec(a)
        if (X >= a)
            log(a)
            X := X - a

module Red
    void log(a)
        if (*)
            dec(a)

Store: X=100                  X=0                  X=-100
       Blue  dec(100)               Red
       Red  log(100)                Blue  dec(100)
       Blue  Callback
       Red  log(100)
module Blue

int X := 100

void dec(a)
    if (X >= a)
        X := X – a
    log(a)

module Red

void log(a)
    if (*)
        dec(a)

Store: X=100 X=0 X=0

Stack/Trace: Blue' dec(100) Blue' dec(100) Red log(100) Red log(100) Blue' dec(100) Callback
Analysis of Ethereum Transactions (7/2015 — 6/2017)

<table>
<thead>
<tr>
<th>Contracts</th>
<th>Invocations</th>
<th>Callbacks</th>
<th>Non-ECF callbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>342,316</td>
<td>96,409,071</td>
<td>284,332</td>
<td>3,312</td>
</tr>
</tbody>
</table>

• Breakdown

  3298: DAO attack
  10: DAO-like vulnerability
  3: Dummy DAO training exercise
  1: DAO demonstration

• Non-ECF callbacks = Attack
  • No “false positive”
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

![Smart Contracts → CERTORA AEV → Verification Report → Test cases to show bugs](image-url)
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 \(Q\)
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 \(Q\)

\[
\text{if } P \text{ then } \{
\text{Contract; assert } Q; \\
\}
\]
Example Hoare Triples

• \{ x=0 \} \ x := x+2 \ \{ x=2 \} \quad \text{✓ valid}

• \{ x=0 \} \ x := x+2 \ \{ x>0 \} \quad \text{✓ valid}

• \{ x=1 \} \ x := x+y \ \{ x>0 \} \quad \times \text{ invalid} \quad \text{Test } y = -3

• \{ x=1 \land y \geq 0 \} \ x := x+y \ \{ x>0 \} \quad \text{✓ valid}

• \{ \text{true} \} \text{ if } x<0 \text{ then } y:=-x \text{ else } y:=x \ \{ y \geq 0 \} \quad \text{✓ valid}

• \{ y \geq 0 \} \ t:=y; \ z:=1; \text{ while } t>0 \text{ do } z:=z*x; \ t:=t-1; \text{ done } \ \{ z=x^y \} \quad \text{✓ valid}
Example Wallet

```c
{ 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 }
```
{ 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 }
{ 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
```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 }
```

```c
{ 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` (Diamond Node)
  - **False** path:
    - `0 < 2`
    - `i := i + 1`
  - **True** path:
    - `{a}`
    - `m_own[own[i]] = true`
    - `i := i + 1`
- `assert count(m_own) = own.len`
wallet_constructor(address[] own)
    int i = 0;
    while (i < own.len)
        m_own[own[i]] = true;
        ++i;
    
    { 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(mOwn) = own.len }
```c
{ 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

True
m_own[own[i]] = true
i := i + 1

False

{ a }
i = 2

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
```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 }
```

```
assert count(m_own)=own.len
```

Diagram:
- **Input:** own=[a,a], m_own={}
  - **Initialization:** i := 0
  - **Loop:**
    - Check: i < own.len
      - If true: m_own[own[i]] = true, i := i + 1
      - If false: i = 2 (assert count(m_own)=own.len)

- **Output:**
  - m_own[own[0]]=true
  - m_own[own[1]]=true
  - i = 2
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 }
\[
\{ \text{count}(m\textunderscore own) = 0 \}
\]

int i = 0;
while (i < own.len) { own.len\geq 0 \land \text{count}(m\textunderscore own) = i } 
if (m\textunderscore own[own[i]])
    abort;
\{ own\textunderscore len\geq 0 \land \text{count}(m\textunderscore own) = i \land \neg m\textunderscore own[own[i]] \} 
m\textunderscore own[own[i]] = true;
\{ own\textunderscore len\geq i \geq 0 \land \text{count}(m\textunderscore own) = i+1 \} 
++i;
\{ own\textunderscore len \geq i \geq 0 \land \text{count}(m\textunderscore own) = i \} 
\{ \text{count}(m\textunderscore own) \geq own\textunderscore 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

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

\[x \neq y\]

Constraint Solver

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