Main difficulty: Some parties can cheat.
Classical result: simulation is possible if the “majority is honest”. For example for 5 players we can tolerate at most 2 “cheaters”.
Formally Verifying Smart Contracts

Mooly Sagiv
Tel Aviv University
And also...
Smart Contracts

• Transactions in bitcoin are limited
  – Transfer ‘X’ bitcoins from ‘Y’ to ‘Z’
• More powerful transactions
  – Exchange
  – Auction
  – Games
  – Bets
  – Legal agreements
• Solution
  – Store smart contracts on the blockchain
  – Computer programs implement transactions
  – Immutability guarantees persistence
Massive Losses due to Bugs

THE PROBLEM
CURRENT SOLUTIONS

Auditing
Testing

Manual
Costly
Incomplete
AUDITING IS INSUFFICIENT

From “A Postmortem on the Parity Multi-Sig Library Self-Destruct”:

… multi-sig wallet code was created and audited by the Ethereum Foundation’s DEV team, Parity technology and others in the community
Automatic software verification

Program P

Desired Property $\varphi$

Solver

Is there a behavior of P that violates $\varphi$?

Y

N

Counterexample

Proof
Disillusionment in program verification 80’s

“Program verification is the holy grail of computer science; always was; always will be”

Challenges in program verification

• Specifying program behavior
• Complexity of program verification
  – The halting problem
  – Rice theorem
  – The ability of simple programs to represent complex behaviors
• Complexity of realistic systems
  – Huge code
  – Heterogeneous code
  – Missing code
The SAT Problem

• Given a propositional formula (Boolean function)
  \(-\varphi = (a \lor b) \land (\neg a \lor \neg b \lor c)\)

• Determine if \(\varphi\) is satisfiable
  – Find a satisfying assignment or report that such does not exist

• For \(n\) variables, there are \(2^n\) possible truth assignments to be checked

• Tools exist: Z3, Yices, CVC, ...

\[\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
\end{array}\]
Verification by reductions to SAT

Program P

Desired Property $\varphi$

Front-End

Formula $[[P]] \land \neg \varphi$

SAT Solver

Counterexample

Proof
Verification by reduction to SAT

SAT Query:

\[((a \land x) \lor (\neg a \land \neg x)) \land ((b \land y) \lor (\neg b \land \neg y)) \land ((x \land \neg y) \lor (\neg x \land y))\]

SAT Answer:
Satisfiable by a=0, b = 1
Verification by reduction to SAT

SAT Query:

\[
((a \land x \land b) \lor (\neg a \land \neg x \land \neg b)) \\
\land \\
((b \land y) \lor (\neg b \land \neg y)) \\
\land \\
((x \land \neg y) \lor (\neg x \land y))
\]

SAT Answer: Unsatisfiable
The SMT (Sat Modulo Theory) Problem

• Given a ground first order formula over theories (Boolean function)
  • \( \varphi = \exists x, y: 2x + y \geq 5 \land y < 3 \)
• Determine if \( \varphi \) is satisfiable
  • Find a satisfying assignment or report that such does not exist
• Satisfiability becomes harder
  • But tools exist: Yices, Z3, CVC, ...
Verification by reductions to SMT

Program P

Desired Property $\varphi$

Front-End

Formula

$[P] \land \neg \varphi$

SMT Solver

Counterexample

Proof
Simple Example Token (buggy)

assert
\[ \forall x. (x \neq \text{to} \land x \neq \text{from}) \Rightarrow b'[x] = b[x] \land 
\text{balance[to]} \geq \text{amount} + \text{fee} \Rightarrow (b'[\text{to}] = \text{balance[to]} - \text{amount} - \text{fee} \land b'[\text{from}] = \text{balance[from]} + \text{amount}) \land 
\text{balance[to]} < \text{amount} + \text{fee} \Rightarrow (b'[\text{to}] = \text{balance[to]} \land b'[\text{from}] = \text{balance[from]}) \]

SMT Answer: Satisfiable by balance[to]=10, fee=5, amount=6, balance[from]=100
Simple Example Token (corrected)

\[ \text{balance}[\text{to}] \geq \text{amount} + \text{fee} \]

\[ \text{balance}[\text{to}] = \text{balance}[\text{to}] - \text{amount} - \text{fee} \]

\[ \text{balance}[\text{from}] = \text{balance}[\text{from}] + \text{amount} \]

assert

\[ \forall x. (x \neq \text{to} \land x \neq \text{from}) \Rightarrow b'[x] = b[x] \land \]
\[ b[\text{to}] \geq \text{amount} + \text{fee} \Rightarrow (b'[\text{to}] = b[\text{to}] - \text{amount} - \text{fee} \land b'[\text{from}] = b[\text{from}] + \text{amount}) \land \]
\[ b[\text{to}] < \text{amount} + \text{fee} \Rightarrow (b'[\text{to}] = b[\text{to}] \land b'[\text{from}] = b[\text{from}]) \]

SAT Answer: Unsatisfiable
More interesting contracts

• Unbounded participants
• Complicated specifications
  • Higher order reasoning
• Need to handle loops
Minting Tokens - buggy

SAT Answer:
Satisfiable by
\( \Sigma \text{balance}=10 \),
\( \text{totalSupply}=10 \),
\( \text{amount}=5 \)

assert

\( \Sigma \text{balance}'=\text{totalSupply}' \wedge \text{totalSupply}'=\text{totalSupply}+\text{amount} \)
Minting Tokens - corrected

$$\Sigma \text{balance} = \text{totalSupply}\ ?$$

- TotalSupply = TotalSupply + amount
- balance[bank] = balance[bank] + amount

assert

$$\Sigma \text{balance}' = \text{totalSupply}' \land \text{totalSupply}' = \text{totalSupply} + \text{amount}$$

SAT Answer: Unsatisfiable
Challenge: Handling Loops

• Bounded loop instantiation
  • CBMC
  • Scaling

• User specified loop invariants
  • Powerful
  • But requires careful insights

• Automatic loop invariants inference
  • Ultimately limited
  • Even when checking is possible

• Limited loops
Summary thus far

• Program verification is powerful
• But hard to apply to complicated systems
• Modularity helps
Runtime Monitoring

- Enforce correctness at runtime
- Especially useful with generic required properties
- Java properties
  - No out of bound array accesses
  - No null dereferences
  - ...
- Can we do the same for contracts?
MOTIVATION: EXISTING VM

Blockchain

DAO

<table>
<thead>
<tr>
<th>Beneficiary</th>
<th>Money</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>2000</td>
</tr>
<tr>
<td>Bob</td>
<td>500</td>
</tr>
<tr>
<td>Thief</td>
<td>0</td>
</tr>
</tbody>
</table>

DAO State:
Balance=0

Withdraw(100$)
Quality VM

Blockchain

Rules

Withdraw(100$)

DAO State:
Balance=2600

<table>
<thead>
<tr>
<th>Beneficiary</th>
<th>Money</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>2000</td>
</tr>
<tr>
<td>Bob</td>
<td>500</td>
</tr>
<tr>
<td>Thief</td>
<td>100</td>
</tr>
</tbody>
</table>

Rule violation
Some Generic Correctness Rules

• Effectively Callback Free (ECF) transactions
  – Eliminate the DAO bug

• Immutable Ownership
  – Parity #1

• Prevent Bad upgrades
  – Monitor code changes and signed whitelists
  – Parity #2

• A flexible framework for arbitrary rules
Effective Callback Freedom – the DAO bug

```cpp
DAO::withdraw(to) {
    if b[to] > 0 {
        sendMoney(to, b[to]);
        b[to] = 0;
    }
}

Thief::uponTransfer(a) {
    DAO::withdraw(Thief)
}

coins[Thief] = 205
b[Thief] = 100
```
EFFECTIVE CALLBACK FREEDOM (ECF)

For every path there is a path without callbacks with same effect
GIST OF DAO ATTACK

For every path there is a path without callbacks with same effect

ECF Could have prevented the DAO bug without human intervention!
Empirical Results (POPL’18)

Ethereum (7/2015 — 6/2017)

<table>
<thead>
<tr>
<th>Blockchain</th>
<th>Contracts</th>
<th>Executions</th>
<th>Non-ECF (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethereum</td>
<td>342K</td>
<td>96M</td>
<td>3,321 (0.003%)</td>
</tr>
<tr>
<td>Ethereum Classic</td>
<td>91K</td>
<td>32M</td>
<td>2,288 (0.007%)</td>
</tr>
</tbody>
</table>

Each Non-ECF is an actual attack (0% False positive)

**Miniscule performance overhead**

**Could have prevented the DAO bug without human intervention!**

*3.38% in time executing EVM alone – drops further in real settings*
THE THREE ENABLERS

• Relatively small number of generic required properties are needed
  • Not per-contract
• Restricted domain
  – Small contracts
  – Modularity due to ECF
• Feasibility of defensive checking

Contract Verification != Software Verification
Complementary Approaches

• Concolic execution
• Restricted programs
Summary

- Virtualization is powerful
- Program verification is powerful
- Program verification is expensive
- Few Success stories
  - Hardware Verification
  - Operating System
  - Device drivers
  - Packet Filters
  - Distributed protocols
- Contract verification
- Higher order programming reduces errors and enables verification