Accurate Smart Contract Verification through Direct Modelling

Matteo Marescotti[1], **Rodrigo Otoni**[1], Leonardo Alt[2],
Patrick Eugster[1], Antti E. J. Hyvärinen[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland
[2] Ethereum Foundation

28th October 2020

Foreword

Industrial application of our work

- Part of the SMTChecker module of the official Solidity compiler
- SMTChecker's constrained Horn clauses model checking engine

Availability of our tool

- `github.com/ethereum/solidity`
- Add `pragma experimental SMTChecker;` to the source file

Formal verification of smart contracts

- `verify.inf.usi.ch/research/fvsc`

Motivation

Why verify smart contracts?

- Smart contracts can hold significant financial assets
- Immutable after deployment
- Source code is publicly available
- Anyone can submit a transaction to a contract

Main approaches used

- Symbolic execution
  - Oyente [Luu et al. CCS'16]
  - MAIAN [Nikolić et al. ACSAC'18]
- Model checking
  - ZEUS [Kalra et al. NDSS'18]
  - SAFEVM [Albert et al. ISSTA'19]
- Interactive theorem proving
  - KEVM [Hildenbrandt et al. CSF'18]

Current Limitation and Proposed Solution

Common feature of existing approaches

- Either imprecise or not fully automated

Imprecision due to translation

- Reuse of established off-the-shelf tools
  - LLVM
  - Boogie
- Need of a translation layer
  - Error prone to develop
  - Requires correctness proofs
  - Adverse effect on precision and efficiency

Our approach

- Direct encoding with native support for **transactionality**
- Solidity as our target language

Example of Transactionality

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```

Example of Transactionality - Initial State

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```

Initialization

State S0

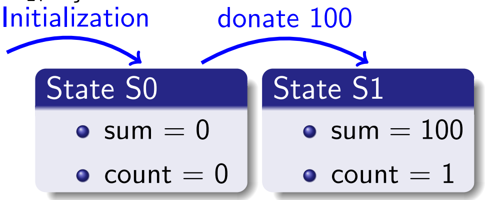
- sum $= 0$
- count $= 0$

Example of Transactionality - `donate.value(100)()`

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```



Initialization

donate 100

State S0

- sum = 0
- count = 0

State S1

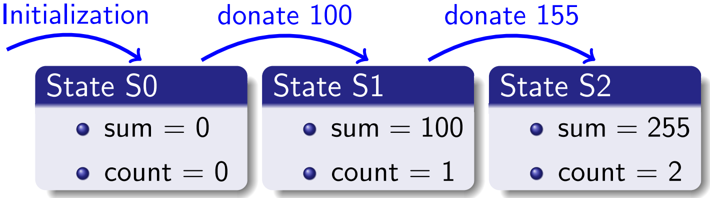
- sum = 100
- count = 1

Example of Transactionality - `donate.value(155)()`

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```



Initialization

donate 100

donate 155

State S0

- $\text{sum} = 0$
- $\text{count} = 0$

State S1

- $\text{sum} = 100$
- $\text{count} = 1$

State S2

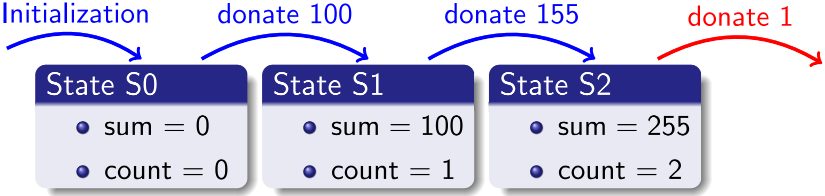
- $\text{sum} = 255$
- $\text{count} = 2$

Example of Transactionality - `donate.value(1)()`

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```



Initialization

State S0
- $\text{sum} = 0$
- $\text{count} = 0$

donate 100

State S1
- $\text{sum} = 100$
- $\text{count} = 1$

donate 155

State S2
- $\text{sum} = 255$
- $\text{count} = 2$

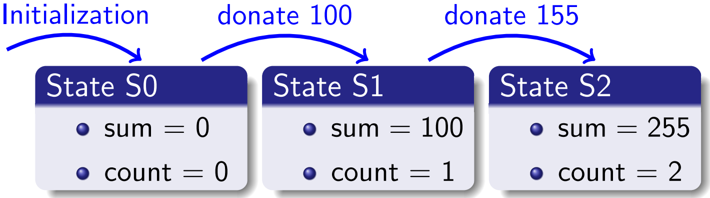
donate 1

Example of Transactionality - `donate.value(1)()`

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```



Initialization

donate 100

donate 155

State S0

- sum $= 0$
- count $= 0$

State S1

- sum $= 100$
- count $= 1$

State S2

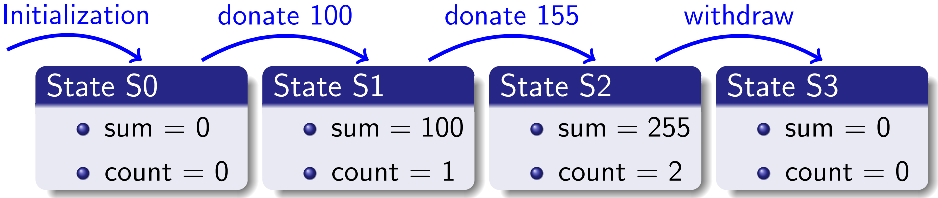
- sum $= 255$
- count $= 2$

Example of Transactionality - `withdraw(0xdCad3a6d...)`

```
contract campaign {
    uint8 sum = 0;
    uint8 count = 0;

    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }

    function withdraw(address receiver) public {
        receiver.transfer(sum);
        sum = 0;
        count = 0;
    }
}
```



Initialization

State S0

- sum = 0
- count = 0

donate 100

State S1

- sum = 100
- count = 1

donate 155

State S2

- sum = 255
- count = 2

withdraw

State S3

- sum = 0
- count = 0

Our Direct Encoding

- Direct encoding based on first-order logic
  - Encoding of smart contracts' control-flow graphs
- Constrained Horn clauses (CHCs)
  - Models the Turing-completeness of smart contracts
  - Used for program verification[1], e.g. C/C++[2] and Java[3] programs

CHC rule format

$$p_1(X_1) \land \ldots \land p_n(X_n) \land \phi \implies h(X)$$

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \land \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \land \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')$$

---

[1] Horn Clause Solvers for Program Verification [Bjørner et al. FLC II'15]

[2] The SeaHorn Verification Framework [Gurfinkel et al. CAV'15]

[3] JayHorn: A Framework for Verifying Java programs [Kahsai et al. CAV'16]

Our Direct Encoding

- Direct encoding based on first-order logic
    - Encoding of smart contracts' control-flow graphs
- Constrained Horn clauses (CHCs)
    - Models the Turing-completeness of smart contracts
    - Used for program verification[1], e.g. C/C++[2] and Java[3] programs

CHC rule format

$$p_1(X_1) \land \ldots \land p_n(X_n) \land \phi \implies h(X)$$

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \land \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \land \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')$$

---

[1] Horn Clause Solvers for Program Verification [Bjørner et al. FLC II'15]

[2] The SeaHorn Verification Framework [Gurfinkel et al. CAV'15]

[3] JayHorn: A Framework for Verifying Java programs [Kahsai et al. CAV'16]

Our Direct Encoding

- Direct encoding based on first-order logic
  - Encoding of smart contracts' control-flow graphs
- Constrained Horn clauses (CHCs)
  - Models the Turing-completeness of smart contracts
  - Used for program verification[1], e.g. C/C++[2] and Java[3] programs

CHC rule format

$$p_1(X_1) \wedge \ldots \wedge p_n(X_n) \wedge \phi \implies h(X)$$

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')$$

---

[1] Horn Clause Solvers for Program Verification [Bjørner et al. FLC II'15]

[2] The SeaHorn Verification Framework [Gurfinkel et al. CAV'15]

[3] JayHorn: A Framework for Verifying Java programs [Kahsai et al. CAV'16]

Our Direct Encoding

- Direct encoding based on first-order logic
  - Encoding of smart contracts' control-flow graphs
- Constrained Horn clauses (CHCs)
  - Models the Turing-completeness of smart contracts
  - Used for program verification[1], e.g. C/C++[2] and Java[3] programs

CHC rule format

$$p_1(X_1) \wedge \ldots \wedge p_n(X_n) \wedge \phi \implies \color{red}{h(X)}$$

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \color{red}{\mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')}$$

---

[1] Horn Clause Solvers for Program Verification [Bjørner et al. FLC II'15]

[2] The SeaHorn Verification Framework [Gurfinkel et al. CAV'15]

[3] JayHorn: A Framework for Verifying Java programs [Kahsai et al. CAV'16]

Our Direct Encoding - Rule Application

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')$$

```
    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }
```

$$\begin{array}{l}
\text{donate1}(\text{sum}, \text{count}, \text{l}_{sum}, \text{l}_{count}, \text{l}_{msg.value}, \text{revert}) \wedge \\
\text{l}_{msg.value} > 0 \wedge \\
\text{l}'_{sum} = \text{l}_{sum} + \text{l}_{msg.value} \wedge \\
\text{l}'_{count} = \text{l}_{count} + 1 \wedge \\
(\text{revert}' = \text{revert} \vee \neg(\text{l}'_{count} \leq \text{l}'_{sum})) \wedge \\
\text{true} \\
\implies \\
\text{donate2}(\text{sum}, \text{count}, \text{l}'_{sum}, \text{l}'_{count}, \text{l}_{msg.value}, \text{revert}')
\end{array}$$

Our Direct Encoding - Rule Application

Rule Jump$_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}')$$

```
    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }
```

$$\begin{aligned}
&\text{donate1}(\text{sum}, \text{count}, \text{l}_{sum}, \text{l}_{count}, \text{l}_{msg.value}, \text{revert}) \wedge \\
&\text{l}_{msg.value} > 0 \wedge \\
&\text{l}'_{sum} = \text{l}_{sum} + \text{l}_{msg.value} \wedge \\
&\text{l}'_{count} = \text{l}_{count} + 1 \wedge \\
&(\text{revert}' = \text{revert} \vee \neg(\text{l}'_{count} \leq \text{l}'_{sum})) \wedge \\
&\text{true} \\
&\implies \\
&\text{donate2}(\text{sum}, \text{count}, \text{l}'_{sum}, \text{l}'_{count}, \text{l}_{msg.value}, \text{revert}')
\end{aligned}$$

Our Direct Encoding - Rule Application

Rule $\mathrm{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}) \land \mathrm{SSA}_{\lambda_v}(\boldsymbol{l},\boldsymbol{l}') \land \mathrm{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}')$$

```
    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }
```

$\mathrm{donate1}(\mathrm{sum}, \mathrm{count}, \mathrm{l}_{sum}, \mathrm{l}_{count}, \mathrm{l}_{msg.value}, \mathrm{revert}) \land$

$\mathrm{l}_{msg.value} > 0 \land$

$\mathrm{l}'_{sum} = \mathrm{l}_{sum} + \mathrm{l}_{msg.value} \land$

$\mathrm{l}'_{count} = \mathrm{l}_{count} + 1 \land$

$(\mathrm{revert}' = \mathrm{revert} \lor \neg(\mathrm{l}'_{count} \leq \mathrm{l}'_{sum})) \land$

true

$\implies$

$\mathrm{donate2}(\mathrm{sum}, \mathrm{count}, \mathrm{l}'_{sum}, \mathrm{l}'_{count}, \mathrm{l}_{msg.value}, \mathrm{revert}')$

Our Direct Encoding - Rule Application

Rule $\text{Jump}_{f,e}$

$$\mathcal{P}^{v}_{f}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l},\boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}^{w}_{f}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}')$$

```
    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }
```

$\text{donate1}(\text{sum}, \text{count}, \text{l}_{sum}, \text{l}_{count}, \text{l}_{msg.value}, \text{revert}) \wedge$
$\text{l}_{msg.value} > 0 \wedge$
$\text{l}'_{sum} = \text{l}_{sum} + \text{l}_{msg.value} \wedge$
$\text{l}'_{count} = \text{l}_{count} + 1 \wedge$
$(\text{revert}' = \text{revert} \vee \neg(\text{l}'_{count} \leq \text{l}'_{sum})) \wedge$
true
$\implies$
$\text{donate2}(\text{sum}, \text{count}, \text{l}'_{sum}, \text{l}'_{count}, \text{l}_{msg.value}, \text{revert}')$

Our Direct Encoding - Rule Application

Rule $\mathrm{Jump}_{f,e}$

$$\mathcal{P}_f^v(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}) \wedge \mathrm{SSA}_{\lambda_v}(\boldsymbol{l},\boldsymbol{l}') \wedge \mathrm{SSA}_{\mu_e}(\boldsymbol{l}') \implies \mathcal{P}_f^w(\boldsymbol{s},\boldsymbol{a},\boldsymbol{l}')$$

```
    function donate() public payable {
        require(msg.value > 0);
        sum = sum + msg.value;
        count = count + 1;
        assert(count <= sum);
    }
```

$$\begin{array}{l}
\mathrm{donate1}(\mathrm{sum},\ \mathrm{count},\ \mathrm{l}_{sum},\ \mathrm{l}_{count},\ \mathrm{l}_{msg.value},\ \mathrm{revert}) \wedge \\
\mathrm{l}_{msg.value} > 0 \wedge \\
\mathrm{l}'_{sum} = \mathrm{l}_{sum} + \mathrm{l}_{msg.value} \wedge \\
\mathrm{l}'_{count} = \mathrm{l}_{count} + 1 \wedge \\
(\mathrm{revert}' = \mathrm{revert} \vee \neg(\mathrm{l}'_{count} \leq \mathrm{l}'_{sum})) \wedge \\
\mathrm{true} \\
\implies \\
\mathrm{donate2}(\mathrm{sum},\ \mathrm{count},\ \mathrm{l}'_{sum},\ \mathrm{l}'_{count},\ \mathrm{l}_{msg.value},\ \mathrm{revert}')
\end{array}$$

Our Direct Encoding - Verification

Assertion checking

- Assertion failures lead to a revert
- Reverts lead to error predicates
- Queries are made for the reachability of error predicates

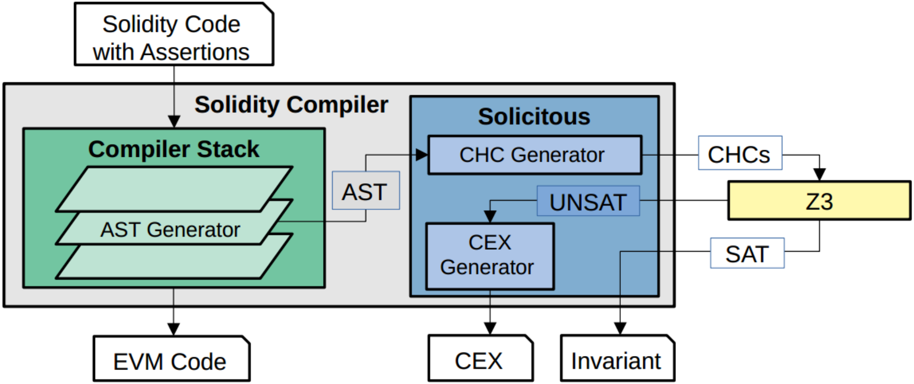
Verification procedure

- System of CHCs represent a contract
- System is provided to a Horn clause solver together with a query
    - Unsatisfiable: an error predicate can be reached
    - Satisfiable: no error predicate can be reached
- Different back-end solvers can be used

Our Tool - Solicitous

SOLIdity Contract verIfication using consTrained hOrn claUSes

- Verification of Solidity contracts
- Applies the encoding rules to generate a system of CHCs
- Generates a contract invariant or a counter-example (CEX)



Experimental Set-up

Benchmarks

- Gathered all deployed contracts between Jan/2019 and May/2020
- Verified 6138 unique real-world contracts containing assertions
  - 77 from version v0.6
  - 6061 from version v0.5

Comparison with existing tools

- Solidity verification tools
  - Solc-Verify [Hajdu et al. VSTTE'19, Hajdu et al. ESOP'20]
  - VeriSol [Wang et al. arXiv'19]
- EVM verification tool
  - Mythril

Experimental Results - v0.5 Contracts

	INT			MOD			
	SOL	SV	VS	SOL	SV	VS	M
Safe	**1720**	778	135	**1681**	54	117	579
Not safe	142	572	298	93	515	198	23
Timeout	586	89	**37**	678	**56**	130	5426
Error	3613	4622	5591	3609	5436	5616	**33**
Verified	**30%**	22%	7%	**29%**	9%	5%	9%

- SOL: Solicitous
- SV: Solc-Verify
- VS: VeriSol
- M: Mythril

- Verified $= \dfrac{\text{Safe} + \text{Not safe}}{\text{Num. of contracts}}$
- Timeout at 60 seconds
- Error means a tool problem

Summary

Problem

- Smart contracts can benefit from formal verification
- Current approaches rely mainly on general encodings

Proposed solution

- Formal verification of safety properties
- Novel approach via direct modelling

Results

- Solicitous, a tool for automatic verification of Solidity contracts
- Improvement in precision when verifying real-world contracts

Future Work

Smart contract verification

- Evaluation of the encoding with languages other than Solidity
- Enhancement of the encoding to account for gas consumption
- Creation of certificates of correctness as witnesses of safe results

Tool improvements

- Implementation of support for additional Solidity features
- Evaluation with different back-end solvers

Formal verification of smart contracts

- `verify.inf.usi.ch/research/fvsc`

Solidity compiler with Solicitous

- `github.com/ethereum/solidity`