SMT-based Compile-time Verification of Safety Properties for Smart Contracts

Leonardo Alt   Christian Reitwiessner

Ethereum Foundation

July 12, 2018

Outline

Outline

Ethereum

- Global networked application platform
- Distributed public database
- Blockchain consensus

Ethereum

- Global networked application platform
- Distributed public database
- Blockchain consensus
- Trustless
- Transparent
- No single entity has control

Smart Contracts

- Accounts controlled by code
- SCs have a storage which can only be written by the SC itself
- The code runs on the Ethereum Virtual Machine (EVM)
- Opcodes have costs (*gas*)
- Transactions may *revert* (exception termination state)

Smart Contracts

- The code is public and immutable after creation
- Monetary incentive to study/attack the program
- The DAO and Parity hacks/accidents

Smart Contracts

- The code is public and immutable after creation
- Monetary incentive to study/attack the program
- The DAO and Parity hacks/accidents

Easy, just use formal verification :)

Smart Contracts - Verification Frameworks

- EVM Formal Semantics: Eth-Isabelle [8], KEVM [9], Ethereum-Lem [10]
- EVM bytecode Symbolic Execution: Oyente [5], Mythril [3], MAIAN [6]
- Translation of Solidity to verifiable languages: Why3 [7], F* [2], ZEUS [4]
- Our approach: Solidity SMT-based BMC
  - Part of the compilation stack
  - No extra effort spent by the developer
  - Counterexamples

Outline

Solidity

- Developed for smart contracts
- Most used language to write smart contracts
- Syntax similar to C/Java
- Main code elements are *contracts*
- Storage and local variables
- Integers of various sizes, *address*, *mapping*, structs, arrays...

Solidity

```
contract Token {
    /// The main balances / accounting mapping.
    mapping(address => uint256) balances;

    /// Create the token contract crediting `msg.sender` with
    /// 10000 tokens.
    constructor() public {
        balances[msg.sender] = 10000;
    }

    /// Transfer `_value` tokens from `msg.sender` to `_to`.
    function transfer(address _to, uint256 _value) public {
        require(balances[msg.sender] >= _value);
        uint256 sumBefore = balances[msg.sender] + balances[_to];
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        uint256 sumAfter = balances[msg.sender] + balances[_to];
        assert(sumBefore == sumAfter);
    }
}
```

Solidity BMC

- Basic idea: SMT-based BMC
- Gas incentive against unbounded loops
- Practicality over Completeness
- Precise encoding

Solidity BMC

Verification targets:

- Underflow / Overflow / Division by 0
- Trivial conditions / Unreachable code
- Assertions

Solidity $\rightarrow$ SMT Encoding

- Traverse the AST
    - Collect constraints
    - Query the SMT solver
- Five types of constraints:
  Control-flow, Type constraint, Variable assignment, Branch conditions, Verification Target

Solidity → SMT Encoding

Branch conditions:

Auxiliary stack that keeps track of the conjunction of conditions that are true at the current program path.
No constraint is added to the solver.

Solidity → SMT Encoding

Control-flow:

Global constraints of the form $b \rightarrow r$, where b is the conjunction of conditions in the current path and r is the condition in `require(r)` and `assert(r)`.

Solidity → SMT Encoding

Type constraint:

Declaration of local variables use default values from types (Integer: 0, Bool: false), and function parameters are initialized with a range of valid values (`uint32 x`: $0 \leq x < 2^{32}$).

Solidity → SMT Encoding

Variable assignment:

The encoding follows the SSA form. When a variable is assigned inside different branches, a new `if-then-else` variable is created after the branching to re-combine the different values.

Solidity → SMT Encoding

Verification target:

- Underflow / Overflow / Division by 0
- Trivial conditions / Unreachable code
- Assertions

Counterexamples are given when a property fails.

Solidity → SMT Encoding

```
contract C
{
  function f(uint256 a, uint256 b)
  {
    if (a == 0)
      require(b <= 100);
    else if (a == 1)
      b = 1000;
    else
      b = 10000;
    assert(b <= 100000);
  }
}
```

1. $a_0 \geq 0 \land a_0 < 2^{256} \land$
2. $b_0 \geq 0 \land b_0 < 2^{256} \land$
3. $(a_0 = 0) \rightarrow (b_0 \leq 100) \land$
4. $b_1 = 1000 \land b_2 = 10000$
5. $b_3 = ite(a == 1, b_1, b_2) \land$
6. $b_4 = ite(a == 0, b_0, b_3) \land$
7. $\neg b_4 \leq 100000$

Outline

Future Plans - Loops

- Automatic detection of loop bounds
- Invariant annotations
- Current syntax:

```
contract C
{
    uint x;
    function f(uint[] _arr) {
        require(_arr.length < 12);
        for (uint i = 0; i < _arr.length; ++i)
            x += _arr[i];
    }
}
```

Future Plans - Multi-transaction Invariants

```
contract C
{
  uint a;

  constructor () public {}

  function a1() public { a = 1; }
  function a2() public { a = 2; }
  function a3() public { a = 3; }
  function a4() public { a = 4; }

  function plusA(uint x) public view returns (uint) {
    require(x < 1000);
    return a + x;
  }
}
```

No transaction leads to $a > 4$

Future Plans - Post-constructor Invariants

```
contract C
{
  uint a;

  constructor (uint x) public {
          require(x < 10);
          a = x;
  }

  function plusA(uint x) public view returns (uint) {
    require(x < 1000);
    return a + x;
  }
}
```

State variable *a* can only be assigned values smaller than 10, and is never assigned again.

Future Plans - Modifiers as pre and postconditions

```
contract C
{
    uint totalSupply;

    modifier safeBalance {
        require(totalSupply == 10000);
        _;
        assert(totalSupply == 10000);
    }

    function transfer(address _to, uint256 _value) safeBalance {
        ...
    }
}
```

Modifier `safeBalance` guarantees that the `totalSupply` does not change after a `transfer`.

Future Plans - Function abstraction

```
contract C
{
    uint totalSupply;

    modifier safeBalance {
        require(totalSupply == 10000);
        _;
        assert(totalSupply == 10000);
    }

    function transfer(address _to, uint256 _value) safeBalance {
        ...
    }

    function zeroAccount(address _to) {
        transfer(_to, balance[msg.sender]);
        assert(totalSupply == 10000);
    }
}
```

The modifier can then be used to abstract function calls and prove properties more efficiently.

Future Plans - Range restriction of real life values

- `block.timestamp` will not exceed 64 bits for the next 500 billion years
- `block.number`

Future Plans - Effective Callback Freeness

- Recently introduced by [1]
- State variable invariants still hold after external calls

Let's build together

github.com/ethereum/solidity

gitter.im/ethereum/solidity-dev

leo@ethereum.org

Thank you!

References I

[1] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. 2(POPL), 48:1–48:28 (2017)

[2] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96. PLAS '16 (2016)

References II

[3] ConsenSys: Mythril (2018), `github.com/ConsenSys/mythril`

[4] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing safety of smart contracts (2018)

[5] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16 (2016)

References III

[6] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. CoRR abs/1802.06038 (2018), `http://arxiv.org/abs/1802.06038`

[7] Why3: Why3 (2018), `why3.lri.fr`

[8] Eth-Isabelle, `github.com/pirapira/eth-isabelle`

[9] KEVM, `github.com/kframework/evm-semantics`

[10] Ethereum-Lem, `github.com/mrsmkl/ethereum-lem`