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

Leonardo Alt and Christian Reitwiessner

Background

Problem

- Smart contract are **immutable** once deployed
- It must be bug-free at deployment time

Suggested Solution (Solidity)

- Compile-time verification
- SMT-based

Require VS. Assert

Similarity – Practical

- Evaluate argument – true/false
- Terminate execution if **false** and revert any previous state changes

Difference - Conceptual

- Require – check **pre**conditions
- Assert – check **post**conditions

Require VS. Assert - Code example

```
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);
}
```

SMT Encoding

Branch conditions

Constraints

- Variable assignment
- Type constraint
- Control-flow

Verification Target

Branch conditions

AST – Abstract syntax tree

- Each if/else statement is a new branch
- Branch conditions – the conditions of the current branch of execution
- Grow and shrink as we traverse the AST
- Let if-statement:

```
if (condition) { << TrueBranch >> }
else { << falseBranch >> }
```

- Add $condition$ to "Branch conditions" during the visit of trueBranch,
  replace with $\neg condition$ during the visit of falseBranch,
  remove that when we are finished with the if-statement.

Constraints

Variable assignment

- SMT variable is assigned only once, SSA – Single Static Assignment
- Each assignment to a program variable introduces a new SMT variable
- Re-combine after condition:

$$var = ite(condition, \quad trueBranchValue, \quad falseBranchValue)$$

```
var = condition ? trueBranchValue : falseBranchValue
```

Constraints

Type constraint

- Variable declaration – default value of the declared type
- Function parameters are initialized with a range of valid values for the given type

Control-flow

- Let b the current Branch conditions state, and r the argument for require(r) or assert(r)
- Add $b \rightarrow r$ to the set of constraints

Verification Target

Arithmetic operations

- Checked against underflow and overflow (according to the type of the values)

Constant branch conditions

- Trivial conditions
- Unreachable blocks

Require & Assert

- Check $\cdots \wedge r$ for require, Unsatisfiable = unreachable code
- Check $\cdots \wedge \neg r$ for assert, Unsatisfiable = assertion failure.

SMT Encoding - example

```
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);
    }
}
```

Type constraint

$a_0 \geq 0 \land a_0 \leq 2^{256}$

$b_0 \geq 0 \land b_0 \leq 2^{256}$

Control-flow

$(a_0 == 0) \rightarrow (b_0 \leq 100)$

Variable assignment

$b_1 = 1000$

$b_2 = 10000$

$b_3 = ite(a_0 == 1, b_1, b_2)$

$b_4 = ite(a_0 == 0, b_0, b_3)$

$\neg(b_4 \leq 100000)$

Future plans

- Multi-transaction invariants
- Function modifiers
- Effective Callback Freeness