Towards the Automated Verification of (Ethereum) Smart Contracts

Chiara Braghin, Elvinia Riccobene and Simone Valentini

chiara.braghin@unimi.it
Why smart contract verification?

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

And...

- Writing code correctly is hard
- Often the semantics of a SC language is not fully understood by programmers
Why smart contract verification?

- DAO Hack on June 17, 2016: worth 3.6 million ETH, about $70 million
- Veritaseum attack on April 2018: worth $8.4 million
- "Double" bZx DeFi Hack:
  - (1) on February 14, 2020 worth $6 million
  - (2) on February 18, 2020, additional $350,000
- Grim Finance on Dec 2021: worth $30 million
- ...
Why Ethereum?
State of Art (1) - Common Vulnerabilities

- Integer Overflow and Underflow
- Default Visibilities
- Race Conditions (Reentrancy, Cross-function race conditions)
- Timestamp Dependence
- DoS with Block Gas Limit
- Forcibly Sending Ether to a Contract
State of Art (2) - Which verification technique?

- Formal verification (of bytecode or Solidity code)
  - Oyente (CCS 2016) – symbolic execution
  - VeriSol (Microsoft 2019) – Boogie intermediate language
  - Solc-Verify (VSSTTE 2019, ESOP 2020)
  - Securify 2.0 (2020) - context-sensitive static analysis in Datalog
  - ....

- Common features of existing formal techniques:
  - Not fully automated
  - Difficult formal languages, translation task might be error prone
  - Not user friendly
  - Not maintained, code not available
  - Focused only on some types of errors/attacks
Our approach: using ASM and ASMETA

- Abstract State Machine (ASM) [1,2]
  - an extension of Finite State Machines, replacing unstructured FSM control states with *algebraic structures*
  - state transitions are performed by firing *transition rules*
  - different *computational paradigms*: single agent and multi-agent
  - ASM model predefined structure: a *signature* with declarations of domains and functions; a block of *definitions* of static domains and functions, transition rules, state invariants and properties to verify; a *main rule*; a set of *initial states*, one of which is elected as default.

Our approach: using ASM and ASMETA

- ASM mETAmodeling: a toolset supporting ASM formal method for model editing, validation and verification [3]

Our approach: using ASM and ASMETA
Our approach: using ASM and ASMETA
Our approach: using ASM and ASMETA

Advantages

- Easy pseudo-code format
- Executable models for different light forms of analysis
- Maintained
contract DAO {
    mapping (address => uint) balances;
    function Deposit() {
        balances[msg.sender] += msg.value;
    }
    function Withdraw(uint amount) {
        if (balances[msg.sender] >= amount) {
            msg.sender.call.value(amount);
            balances[msg.sender] -= amount;
        }
    }
}

balances is updated only after ether transfer
ASM by example (1)

contract DAO {
    mapping (address => uint) balances;
    function Deposit() {
        balances[msg.sender] += msg.value;
    }
    function Withdraw(uint amount) {
        if (balances[msg.sender] >= amount) {
            msg.sender.call.value(amount);
            balances[msg.sender] -= amount;
        }
    }
}

balances is updated only after ether transfer

contract Attacker {
    ...
    function moveBalance() {
        dao.Withdraw();
    }
    function () payable {
        dao.Withdraw();
    }

ASM by example (2)

Set-up

```c
enum domain ExecutingFunction = {DEPOSIT, WITHDRAW, CALLBACK, NONE}
enum domain User = {USER, DAO, ATTACKER}
enum domain Contract = {CONTRACTDAO, CONTRACTATTACKER}
domain MoneyAmount subsetof Integer
domain StackLayer subsetof Integer
domain InstructionPointer subsetof Integer
domain GeneralInteger subsetof Integer

dynamic controlled balance : User -> MoneyAmount
derived user_contract : User -> Contract
derived is_contract : User -> Boolean

/* FUNCTIONS THAT ALLOW TRANSACTIONS */
controlled sender : StackLayer -> User
dynamic controlled amount : StackLayer -> MoneyAmount

/* STACK MANAGEMENT */
dynamic controlled current_layer : StackLayer

/* ALLOW FUNCTION EXECUTIONS */

/* State Functions */
dynamic controlled executing_function : StackLayer -> ExecutingFunction
dynamic controlled instruction_pointer : StackLayer -> InstructionPointer
dynamic controlled executing_contract : StackLayer -> Contract

/* CONTRACT ATTRIBUTES, PARAMETERS AND RETURN VALUES */
/* CONTRACT ATTRIBUTES */
dynamic controlled customer_balance : User -> MoneyAmount
dynamic controlled counter : GeneralInteger

/* FUNCTIONS PARAMETERS */
controlled value_deposit : StackLayer -> MoneyAmount // amount of coin to deposit calling
controlled value_withdraw : StackLayer -> MoneyAmount // amount of coin to withdraw calling

/* definitions: */
```

ASM by example (2)

```plaintext
/* TRANSACTION RULE */

rule r_Transaction($s in User, $r in User, $n in MoneyAmount, $f in ExecutingFunction) =
  if balance($s) >= $n and $n >= 0 then
    let ($ucr = user_contract($r), $cl = current_layer) in
      par
        balance($s) := balance($s) - $n // subtracts the amount from the sender user balance
        balance($r) := balance($r) + $n // adds the amount to the dest user balance
      endpar
    if is_contract($r) then
      sender($cl + 1) := $s // set the transition attribute to the sender user
      amount($cl + 1) := $n // set the transaction attribute to the amount of coin to transfer
      current_layer := $cl + 1
      executing_contract($cl + 1) := $ucr
      executing_function($cl + 1) := $f
      instruction_pointer($cl + 1) := 0
      instruction_pointer($cl) := instruction_pointer($cl) + 1
    endif
  endif
endlet

/* RETURN RULE */

rule r_Ret =
  current_layer := current_layer - 1
```
function Deposit() {
    balances[msg.sender] += msg.value;
}

function Withdraw(uint amount) {
    if (balances[msg.sender] >= amount) {
        msg.sender.call.value(amount);
        balances[msg.sender] -= amount;
    }
}
ASM by example (2)

```plaintext
/*
 * CALLBACK FUNCTION RULE
 */

rule r_Callback_bank1 =
  let ($cl = current_layer) in
  if executing_function($cl) != DEPOSIT and executing_function($cl) != WITHDRAW then
    switch instruction_pointer($cl)
    case 0 :
      r_Ret[]
      endif
  endlet

/*
 * ATTACKER CONTRACT IMPLEMENTATION
 */

/*
 * CALLBACK FUNCTION RULE
 */

rule r_Callback_bank2 =
  let ($cl = current_layer) in
  switch instruction_pointer($cl)
  case 0 :
    if counter < 2 then
      par
      r_Transaction[ATTACKER, DAO, 1, WITHDRAW]
      counter := counter + 1
      endpar
    else
      r_Transaction[ATTACKER, DAO, 1, CALLBACK]
    endif
  case 1 :
    r_Ret[]
  endlet

/*
 * MAINS AND INVARIANTS
 */
```
**ASM by example (3)**

```plaintext
VALIDATION
```

---

```plaintext
/**
 * MAIN AND INvariants
 */

/*
 * invariant over costumer_balance : costumer_balance(ATTACKER) >= 0
 */

/*
 * CTLSPEC
 */

CTLSPEC ef(costumer_balance(ATTACKER) < 0)
```
VERIFICATION

```plaintext
/* invariant */

invariant over costumer_balance : costumer_balance(ATTACKER) >= 0

/* */

/* CTLSPEC */

CTLSPEC ef(costumer_balance(ATTACKER) < 0)
```
Future Works

• Generalize the approach
  - Model the full semantics of SC
  - Identify common patterns in the structure of the contract to build an ASM library
  - Build a catalog of common vulnerabilities and express it in terms of properties to check
  - A GUI for specification of verified smart contracts?
  - What about other blockchains?