Analysis and Optimization of (Ethereum) Smart Contracts

Elvira Albert,

Jesús Correas, Pablo Gordillo, Clara Rodríguez-Nuñez, Guillermo Román-Díez and Albert Rubio

FORTE-CM Annual Meeting

Universidad Complutense de Madrid (Spain)

December 2019

Introduction (Ethereum)

- Ethereum is a global, open-source platform for decentralized applications that has become the world's leading programmable blockchain.

Introduction (Ethereum)

- Ethereum is a global, open-source platform for decentralized applications that has become the world's leading programmable blockchain.
- Ethereum has a native cryptocurrency named *Ether*.

Introduction (Ethereum)

- Ethereum is a global, open-source platform for decentralized applications that has become the world's leading programmable blockchain.
- Ethereum has a native cryptocurrency named *Ether*.
- Ethereum is programmable using a Turing complete language in which developers can code smart contracts that control digital value, run exactly as programmed, and are immutable.

Introduction (Ethereum)

- Ethereum is a global, open-source platform for decentralized applications that has become the world's leading programmable blockchain.
- Ethereum has a native cryptocurrency named *Ether*.
- Ethereum is programmable using a Turing complete language in which developers can code smart contracts that control digital value, run exactly as programmed, and are immutable.
- A *smart contract* is basically a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Introduction (Ethereum)

- Ethereum is a global, open-source platform for decentralized applications that has become the world's leading programmable blockchain.
- Ethereum has a native cryptocurrency named *Ether*.
- Ethereum is programmable using a Turing complete language in which developers can code smart contracts that control digital value, run exactly as programmed, and are immutable.
- A *smart contract* is basically a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.
- Why interesting for the Formal Methods community?
  - They are relatively small in size
  - They are very valuable (in *Ether*)
  - They require proving new properties

Introduction (EVM)

- *Solidity* is a programming language to write smart contracts for the Ethereum platform
  - Solidity programs are compiled into `EVM` (Ethereum Virtual Machine) bytecode
  - The EVM is published and used in the Ethereum platform

Introduction (EVM)

- *Solidity* is a programming language to write smart contracts for the Ethereum platform
  - Solidity programs are compiled into `EVM` (Ethereum Virtual Machine) bytecode
  - The EVM is published and used in the Ethereum platform
- The fee of executing a transaction in the Ethereum platform is measured in units of *gas*:

Introduction (EVM)

- *Solidity* is a programming language to write smart contracts for the Ethereum platform
  - Solidity programs are compiled into `EVM` (Ethereum Virtual Machine) bytecode
  - The EVM is published and used in the Ethereum platform
- The fee of executing a transaction in the Ethereum platform is measured in units of *gas*:
  - Each EVM bytecode instruction consumes gas for executing
  - If a transaction exceeds the amount of gas allotted by the user (known as gas limit), *out-of-gas* exception
  - Miners paid an amount in *Ether* equivalent to the total amount of gas it took them to execute a complete operation.

Introduction (EVM)

- *Solidity* is a programming language to write smart contracts for the Ethereum platform
  - Solidity programs are compiled into `EVM` (Ethereum Virtual Machine) bytecode
  - The EVM is published and used in the Ethereum platform
- The fee of executing a transaction in the Ethereum platform is measured in units of *gas*:
  - Each EVM bytecode instruction consumes gas for executing
  - If a transaction exceeds the amount of gas allotted by the user (known as gas limit), *out-of-gas* exception
  - Miners paid an amount in *Ether* equivalent to the total amount of gas it took them to execute a complete operation.
- `EVM` includes a bytecode instruction (`INVALID`) to abort a transaction avoiding incorrect executions
  - It reverts the state of the contract
  - The initial *gas* of the transaction is not refunded and, hence, its execution has economical consequences

Research Lines (COSTA Group)

1. Estimate the gas consumption (static analysis)
   - Goal: avoid gas-related vulnerabilities
2. Optimize the gas consumption (synthesis)
   - Goal: reduce transaction costs
3. Prove safety (static analysis)
   - Goal: ensure correctness of executions (`INVALID` not reachable)
4. Prove the ECF property (dynamic and static analysis)
   - Goal: ensure no reentrancy attacks (`DAO` bug)

1. Estimate the gas consumption

GASTAP

GASTAP is a framework that given a smart contract infers gas upper bounds for all its public functions

```
contract EthereumPot {
  address public owner;
  address[] public addresses;
  address public winnerAddress;
  uint[] public slots;
  ...
  function findWinner(uint random) constant returns(address winner){
    for(uint i = 0; i < slots.length; i++) {
      if(random <= slots[i]) {
        return addresses[i];
      }}
  }
  ...
}
```

1. Estimate the gas consumption

GASTAP

GASTAP is a framework that given a smart contract infers gas upper bounds for all its public functions

```
contract EthereumPot {
  address public owner;
  address[] public addresses;
  address public winnerAddress;
  uint[] public slots;
  ...
  function findWinner(uint random) constant returns(address
    for(uint i = 0; i < slots.length; i++) {
      if(random <= slots[i]) {
        return addresses[i];
      }}
  }
  ...
}
```

15+ (1555+779·slots)

Is it more difficult than resource analysis?

- YES: the gas cost depends on the opcodes executed and the memory used during the transaction
  - Memory Gas
  - Opcode Gas

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

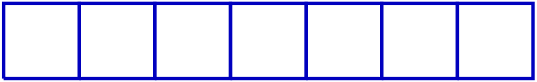
- i is a program point
- $C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$
- μ_i' and μ_i are the highest memory slot accessed after and before the execution of the opcode at program point i

Is it more difficult than resource analysis?

Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu'_i)\text{-}C_{mem}(\mu_i) + C_{opcode}(i)$$

$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

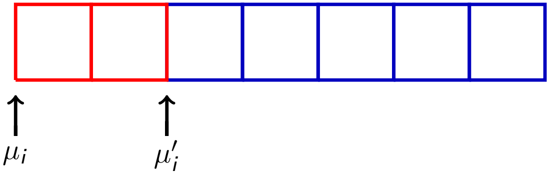


Is it more difficult than resource analysis?

Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu_i') - C_{mem}(\mu_i) + C_{opcode}(i)$$

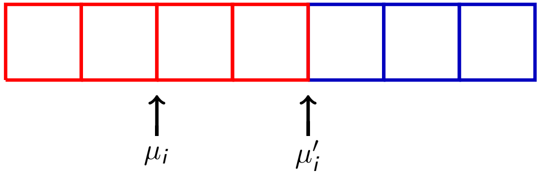
$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$



Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu'_i) - C_{mem}(\mu_i) + C_{opcode}(i)$$

$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

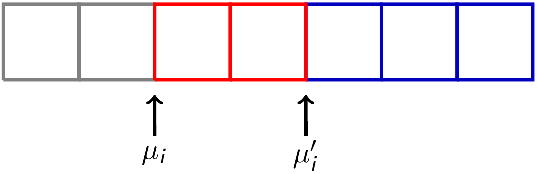


Is it more difficult than resource analysis?

Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu'_i) \text{-} C_{mem}(\mu_i) + C_{opcode}(i)$$

$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$



Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

The sum of all memory gas cost $\equiv$ the memory cost function for the highest slot accessed

Memory Gas

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i) + C_{opcode}(i)$$

$$C_{mem}(a) = 3 \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

The sum of all memory gas cost $\equiv$ the memory cost function for the highest slot accessed

- Infer the current highest slot accessed by any operation in the function

Is it more difficult than resource analysis?

Opcode Gas

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i) + C_{opcode}(i)$$

- Each opcode has a fee associated
  - Most of them are constant: JUMP(2), ADD(3), MLOAD(3), SLOAD(200), etc.
  - Different constant gas depending on some condition: SSTORE (20000 / 5000), etc.
  - Non-constant gas consumption: EXP, SHA3, etc.

Is it more difficult than resource analysis?

Opcode Gas

- **Constant gas**

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

They can be classified in 8 groups:

- Zero (0): STOP, RETURN, REVERT
- Base (2): POP, GAS, CALLER, ADDRESS,...
- VeryLow (3): ADD, SUB, LT, EQ, PUSH, MLOAD, MSTORE,...
- Low (5): MUL, DIV, MOD, SMOD,...
- Mid (8): ADDMOD, MULMOD, JUMP
- High (10): JUMPI
- Extcode (700): EXTCODESIZE
- Others: JUMPDEST(1), SLOAD(200), CREATE(32000),...

Opcode Gas

- **Constant gas depending on some condition**

$$\text{Gas(i)} = C_{mem}(\mu'_i)\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

When storing value v in address pos:

$$\textbf{SSTORE} = \begin{cases} 20000 & \text{if} \quad storage(pos) == 0 \wedge v \neq 0 \\ 5000 & \text{otherwise} \end{cases}$$

Opcode Gas

- **Constant gas depending on some condition**

$$\text{Gas(i)} = C_{mem}(\mu_i')\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

$$\textbf{CALL} = 700 + C_{\text{XFER}} + C_{\text{NEW}}$$

$$C_{\text{XFER}} \rightarrow \begin{cases} 9000 & \text{if stack[top-2]} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$C_{\text{NEW}} \rightarrow \begin{cases} 25000 & \text{if the contract doesn't exist} \\ 0 & \text{otherwise} \end{cases}$$

Is it more difficult than resource analysis?

Opcode Gas

- **Non-constant gas**

$$\text{Gas(i)} = C_{mem}(\mu'_i)\text{-}C_{mem}(\mu_i)+C_{opcode}(i)$$

- $\text{EXP} = 10 + 50 \cdot (1 + \lfloor log_{256}(stack[top - 1]) \rfloor)$
- $\text{SHA3} = 30 + 6 \cdot \lceil stack[top - 1] \div 32 \rceil$
- $\text{LOG3} = 375 + 8 \cdot stack[top - 1] + 3 \cdot 375$
- $\cdots$

Gas estimation of Ethereum smart contracts

1. The gas model of Ethereum is complex
   - New components to estimate: highest slot accessed
   - Non-constant costs
2. A resource analysis problem
   - bound number of iterations
   - worst-case approximations

2. Optimization of gas consumption

- Executing EVM byte-code is subject to monetary fees: a clear optimization target.
- Our approach: use the gas analysis to detect gas-expensive fragments of code and automatically optimize them.
- Storage optimization: (global) write access costs 20.000 in the worst case and 5.000 in the best case while local memory costs only 3
- Goal: replace *multiple* accesses to the same (global) storage data by *one* access that copies the data in storage to a (local) memory position followed by accesses to such memory position and a *final* update to the storage if needed.

2. Optimization of gas consumption

Original program:

```
for (uint i=0; i<data.length; i++) {
   address a = address( data[i] & (D160-1) );
   uint amount = data[i] / D160;
   if (balanceOf[a] == 0) {
     balanceOf[a] = amount;
     totalSupply += amount;
   }
 }
```

2. Optimization of gas consumption

Original program:

```
for (uint i=0; i<data.length; i++) {
   address a = address( data[i] & (D160-1) );
   uint amount = data[i] / D160;
   if (balanceOf[a] == 0) {
     balanceOf[a] = amount;
     totalSupply += amount;
   }
 }
```

Optimized program:

```
 uint256 totalSupply = get_field_totalSupply();
  for (uint i=0; i<data.length; i++) {
   address a = address( data[i] & (D160-1) );
   uint amount = data[i] / D160;
   if (balanceOf[a] == 0) {
     balanceOf[a] = amount;
     totalSupply += amount;
   }
 }
set_field_totalSupply(totalSupply);
```

2. Optimization of gas consumption

What are the difficulties?

- Detect the pattern
  - ensure effectiveness
- Guarantee soundness
  - global data not reachable by transitive calls
  - analysis that can be done with different precision levels
- Implementation
  - analysis performed on the EVM
  - changes made in the Solidity

3. Prove safety of Solidity programs

Safety: ensure a correct behaviour of EVM code

- **Array accesses**: when an array access is done out of the bounds of the array an `INVALID` is executed

  `a[i]` $\Rightarrow$ `if (i >= a.length) then INVALID`

- **Assert statements**: checks the condition and invokes `INVALID` when the condition does not hold

  `assert(boolexp)` $\Rightarrow$ `if (!boolexp) then INVALID`

- **Divisions by zero**: `INVALID` is executed to avoid divisions by zero

  `a/b` $\Rightarrow$ `if (b == 0) then INVALID`

- **Enumerate types**: EVM checks that the number representing an enumerate element is within the range of the enumeration and executes `INVALID` when it is not in it (e.g. it is received as parameter)

What is safevm?

SAFEVM

SAFEVM is a **verification tool** that takes as input a function of a Ethereum smart contract, transforms it into a C program, and, using a verification engine, produces a verification result for guaranteeing the unreachability of the `INVALID` operations

Smart Contract → CFG + RBR → C Program → Verification Result

Conclusions and future work

- Analysis and verification of Ethereum smart contracts is becoming a popular research topic
  - There are tools based on symbolic execution, SMT solving or certified programming for detecting safety and security vulnerabilities
- Our tools:
  - verify safety of EVM code
  - estimate the gas fee for running transactions
  - certify that the contract is free of out-of-gas vulnerabilities
  - attackers: estimate how much *Ether* an adversary has to pour into a contract in order to execute an out-of-gas attack
  - the gas analysis can be used to detect gas-expensive fragments of code and automatically optimize them.