FORTE-CM Summer School
July 2022

Intro to Smart Contracts, Ethereum and Solidity

Pablo Gordillo – pabgordi@ucm.es
Jesús Correas – jcorreas@ucm.es

Universidad Complutense de Madrid

Blockchain Systems

What is a blockchain?

A blockchain is an implementation of a **decentralized ledger**, which is a simple **database of transactions** of *coins* (in some digital currency) between addresses.

- Main characteristics:
  - **immutable:** cryptographic techniques prevent the ledger from being modified.
  - **verifiable:** anyone can check that transactions are correct.
  - **decentralized:** processed by *miners* (or *validators*).
- The blockchain technology allows to register information more general than a ledger for payments.

Notion of smart contract

- In particular, blockchain technology can be used **to automate some rules usually contained in legal contracts or agreements.**
  - If the blockchain system can guarantee by construction the transactions without any intermediator, then
  - it could also be used for **automatically executing** the terms of a contract.
- The term **smart contract** was coined by Nick Szabo, a computer scientist and legal scholar, before blockchain technology was developed:

> A **smart contract** is a set of promises, specified in digital form, including protocols within which the parties perform on these promises.[1]

[1] N. Szabo. Smart Contracts: Building Blocks for Digital Markets, 1996

Ethereum

- A full-fledged computer language would be **very appropriate for implementing smart contracts.**

> Contracts should be executed in a trustworthy environment such as **a blockchain system.** This is the case of **Ethereum.**

- **Ethereum** is more than just another blockchain system.
- The main goal of Ethereum is to create a protocol for building decentralized applications.
  - It is able to **execute bytecode in a built-in virtual machine.**
  - **Ether currency** is used to pay for the execution of contracts and to reward miners for mining new blocks.
- The state in Ethereum is composed of a set of **accounts:**
  - **Externally owned accounts (EOA).**
  - **Contract accounts:** Accounts controlled by a smart contract.

Ethereum Contracts

"A contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. Contract accounts are able to pass messages between themselves as well as doing practically Turing complete computation. Contracts live on the blockchain in a Ethereum-specific binary format called Ethereum Virtual Machine (EVM) bytecode."

From `https://ethdocs.org/en/latest/contracts-and-transactions/contracts.html`

- Code and data **reside safe on the Ethereum blockchain.**
- **Turing complete computation:** it allows any arbitrary computation.
- **Ethereum Virtual Machine (EVM):** Contracts **are compiled** into a hardware-independent bytecode language.
- **Contracts live on the blockchain:** executed in a **secured environment,** in the network nodes.

Ethereum Virtual Machine

- Ethereum includes a virtual machine that executes bytecode (very much like Java VM).
- **EVM** bytecode is a platform-independent assembly-like language.
- High-level languages compile into EVM bytecode, e.g. **Solidity.**
- An important problem of a programmable blockchain are **infinite computations.**
  - They can be used as a form of DDoS attack.
  - They cannot be checked in advance (unless we manage to solve the **halting problem!**).
- This problem is overcome in Ethereum with the notion of **gas.**

The notion of *gas*

- **DoS attacks and infinite computations** are prevented by **charging** the execution of contract bytecode with a fee:

> **Every EVM instruction consumes some amount of gas.**

- Some instructions are more expensive than others (e.g., persistent storage handling).
- Local memory usage also consumes gas.
- When a contract function is invoked, two values are provided:
  - **gasLimit:** the amount of gas available for executing the contract.
  - **gasPrice:** the value of each unit of gas measured in Wei.
- If gasLimit is **exceeded**, the execution halts with an **exception** and **the entire execution is reverted.**

Solidity

- **Solidity** is the most commonly used language for programming contracts in Ethereum.
- The **syntax of Solidity** is similar to other well-known programming languages (C, Java).
- However, it includes many details imposed by the **low-level EVM** bytecode language.
- **Contracts** are **similar to classes** in object-oriented languages.
- Contracts can contain the following elements (among others):
  - **State variables:** persistent variables that are stored on Ethereum blockchain **storage.**
  - **Functions:** define the behaviour of the contract.
  - **Events:** log entries for communicating info to external applications.

Layout of a Solidity source file

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract Storage {
  uint number;  // these are (persistent) state variables
  address owner;

  function store(uint num) public {
    uint double; // this is a local variable
    double = num * 2;
    number = double;
  }

  function retrieve() public view returns (uint){
    return number;
  }
}
```

Solidity Data Types

- **Value data types:**
  - **`uint`**, **`uint256`**: 256-bit unsigned integers.
  - **`int`**, **`int256`**: 256-bit signed integers.
  - **`uint8`**, **`int8`**, **`uint16`**, **`uint32`**,...: (un)signed integers of N bits.
  - **`address`**: 160-bit Ethereum addresses.
- **Reference data types:**
  - **structs**
  - **arrays** (fixed- and dynamically-sized)
  - **mappings** (similar to hash tables)

FORTE-CM Summer School
July 2022

EVM programs

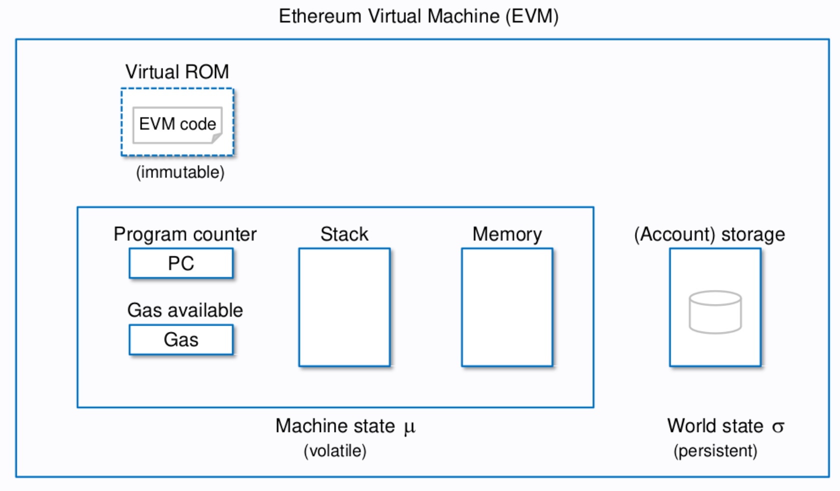
Pablo Gordillo – pabgordi@ucm.es
Jesús Correas – jcorreas@ucm.es

Universidad Complutense de Madrid

Ethereum Bytecode (EVM)

- Stack based language
- Small set of instructions
  - Arithmetic operations
  - Conditional and unconditional jumps
  - Store and load data
  - Blockchain data
  - Cryptographic hash algorithms for allocating persistent memory
- A **specification of the EVM** is available:
  G. Wood, *Ethereum: A secure decentralised generalised transaction ledger (EIP-150)*, 2018.
  `https://ethereum.github.io/yellowpaper/paper.pdf` (latest version)

EVM architecture



Source: `https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

EVM bytecode

- Bytecode language is untyped.
  - Most instructions operate on 256-bit words as unsigned integers.
  - Specific signed operations (`SDIV`, `SMOD`, `SIGNEXTEND`).
- All invocations start executing the contract with PC $= 0$.
- Runtime code starts with some memory initializations and the **function selector.**
  - A function is identified by a hash of its signature:
  - Example: `bytes4(keccak256(bytes("square(uint256)")))`
- There is no notion of methods, objects or structs.
- Calls to internal functions are **translated to jumps.**
- Jump addresses are not constant. They are read from the stack.

EVM calls to internal functions

```
pragma solidity ^0.6.0;

contract mini2 {
  function square(uint n)
  external pure returns(uint) {
    return _square(n);
  }

  function _square(uint n)
  internal pure returns(uint) {
    return n*n;
  }
}
```

```
   ...
   JUMPDEST
   PUSH 0
   PUSH 8 //tag8
   DUP3
   PUSH 9 //tag9
   JUMP
tag8
   JUMPDEST
   SWAP1
   POP
   SWAP2
   SWAP1
   POP
   JUMP //out
```

```
tag9 //_square
   JUMPDEST
   PUSH 0
   DUP2
   DUP3
   MUL
   SWAP1
   POP
   SWAP2
   SWAP1
   POP
   JUMP //out
```

EVM bytecode instructions

- **Arithmetic:**
  `ADD`, `MUL`, `DIV`, `SDIV`, `MOD`, `SMOD`, `ADDMOD`, `MULMOD`, `EXP`, `SIGNEXTEND`
- **Comparison and bitwise logic:**
  `LT`, `GT`, `SLT`, `SGT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, `NOT`, `BYTE`, `SHL`, `SHR`, `SAR`
- **Environmental information:** `ADDRESS`, `BALANCE`, `ORIGIN`, `CALLER`, etc.
- **Stack:** `POP`, `PUSH`i, `DUP`j, `SWAP`j ($i <= 32, j <= 16$)
- **Memory:** `MLOAD`, `MSTORE`
- **Storage:** `SLOAD`, `SSTORE`
- **Block information:** `BLOCKHASH`, `COINBASE`, `GASLIMIT`, etc.
- **Flow:** `JUMP` (unconditional jump),`JUMPI` (conditional jump), `JUMPDEST` (jump destination)
- **Hash computation:** `SHA3`
- **System:** `CREATE`, `CALL`, `DELEGATECALL`, `RETURN`, `REVERT`, `STOP`, `INVALID`, `SELFDESTRUCT`, etc.

EVM bytecode

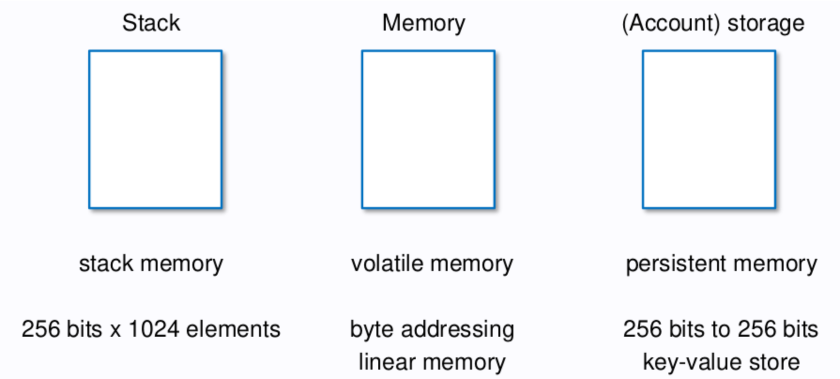
Notion of *block*

"A maximal sequence of straight-line consecutive code in the program with the properties that the flow of control can only enter the block through the first instruction in the block, and can only leave the block at the last instruction."

- Blocks are the **nodes** in the **control-flow graph (CFG)** of the contract.

EVM memory regions

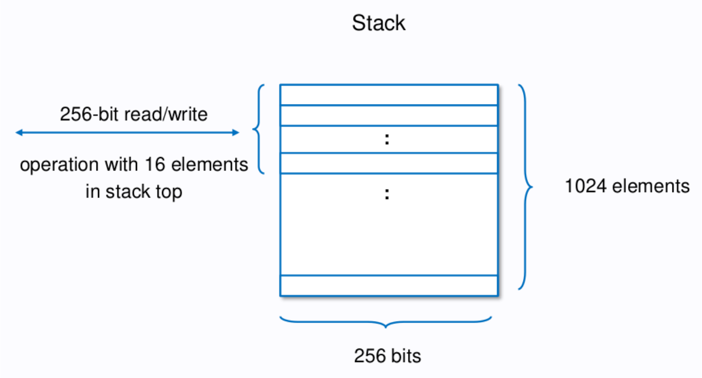
- There are several memory regions:



Source: `https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

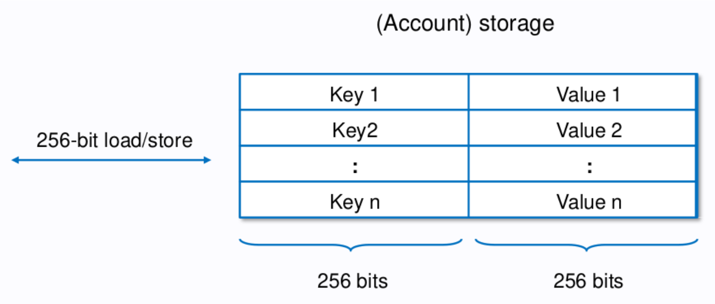
EVM stack

- EVM is a **stack-based virtual machine:**
  - There are no registers, just a stack with 1024 256-bit words.
  - Most instructions operate on the top-most elements in the stack.
  - Contains **value-type local variables,** intermediate values and **jump addresses.**
  - **Gas consumption:** Operations on stack are **very cheap.**



EVM storage

- **Persistent memory** region.
- Each contract can only access its own storage.
- Implemented as a collection of (key,value) pairs.
- Stores any data type: **value types, structs, arrays, mappings.**
- **Gas consumption:** Expensive to use (the price of persistence).

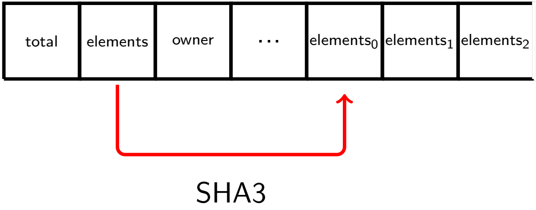


Source: `https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

Storage layout

- State variables are stored in consecutive positions starting at 0.
- Value types are stored directly, packing them tightly if possible.
- Arrays are stored differently:
  - Array length is stored in its corresponding slot.
  - Array contents are stored at location SHA3(slot).

Storage:



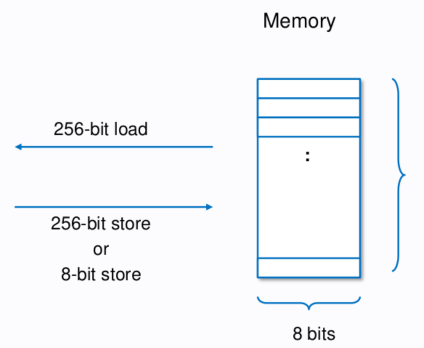
Storage operations

```
contract StAccesses {
  uint total;
  uint[] elems;
  address owner;

  function fn(uint n) external {
    for (uint i=0; i<n; i++){
      elems.push(0x97);
    }

    total = 13;
    owner = msg.sender;
  }
}
```

```
...
SLOAD      //elems.length
ADD        //length+1
DUP1
DUP3
SSTORE     //elems.length
...
KECCAK256 //addr(elems[0])
ADD   //addr(elems[length-1])
...
SSTORE     //elems[length-1]
...
PUSH D
PUSH 0
DUP2
SWAP1
SSTORE    //store total
...
```

EVM memory

- Volatile memory region for local computations:
  - Fresh memory for each external function call.
  - Stores **reference-type local variables** (structs, arrays).
  - Required by some instructions (`SHA3`, arguments of external calls).
  - **Gas consumption:** Cheaper than storage, but gas increases with the amount of memory used.



Source: `https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf`

Memory layout

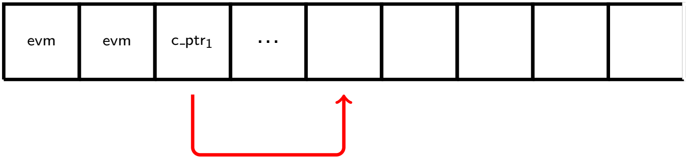
- Memory address `0x40` contains a pointer to the first free memory location.
  - It is increased for each variable.
  - Each basic value is stored in a 256-bit word (regardless of its size).
- It is **never freed.** Cleared for each external call.
- Arrays are stored next to their length.
- Pointers are used for nested data structures.

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint[](7);
    //...
  }
}
```

Memory layout

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint8[](7);
    //...
  }
}
```

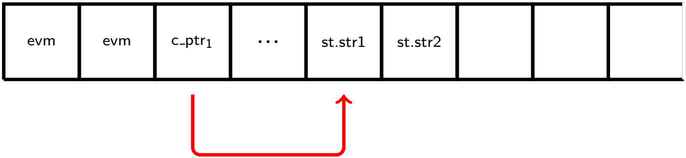
Memory:



Memory layout

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint8[](7);
    //...
  }
}
```

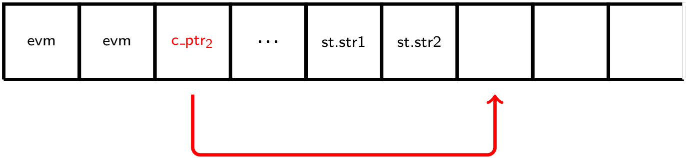
Memory:



Memory layout

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint8[](7);
    //...
  }
}
```

Memory:



Memory layout

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint8[](7);
    //...
  }
}
```

Memory:

evm	evm	c_ptr_2	$\cdots$	st.str1	st.str2	len(arr1)	$arr1_2$	$arr1_2$

Memory layout

```
contract Accesses {
  struct StrType { uint str1; uint str2; }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint8[](7);
    //...
  }
}
```

Memory:

evm	evm	c_ptr_3	$\cdots$	st.str1	st.str2	len(arr1)	$arr1_2$	$arr1_2$

Memory operations

```
contract Accesses {
  struct StrType {
    uint str1;
    uint str2;
  }

  function fnMemory() external pure  {
    StrType memory st=StrType(15,11);
    uint[] memory arr1=new uint[](7);

    //...
  }
}
```

```
PUSH 40
DUP1
MLOAD   // ptr=mem[0x40]
SWAP1
DUP2
ADD
PUSH 40
MSTORE //mem[0x40]=ptr+0x40
DUP1
PUSH F
DUP2
MSTORE  //mem[ptr] = 15
PUSH 20 // (st.str1)
ADD
PUSH B
DUP2
MSTORE  //mem[ptr+0x20]=11
...     // (st.str2)
```

FORTE-CM Summer School
July 2022
Analysis of Smart Contracts
Gas Analysis of Ethereum Smart Contracts

Pablo Gordillo – pabgordi@ucm.es
Jesús Correas – jcorreas@ucm.es

Universidad Complutense de Madrid

+Efficient Smart Contracts

Efficiency-related properties:

1. Estimate the gas consumption (static analysis)
   - Goal: avoid gas-related vulnerabilities (**GASTAP**)
2. Optimize the gas consumption (synthesis)
   - Goal: reduce transaction costs (**GASOL**)

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

 300.000

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

300.000 $\longrightarrow$ $\boxed{\text{21.000 + 25.000 execution}}$

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

300.000 $\rightarrow$ $\boxed{21.000 + 25.000 \text{ execution}}$ $\rightarrow$ 257.000

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

 300.000 → $\boxed{\text{21.000 + 25.000 execution}}$  → 257.000 

 30.000

Gas-Metered Execution

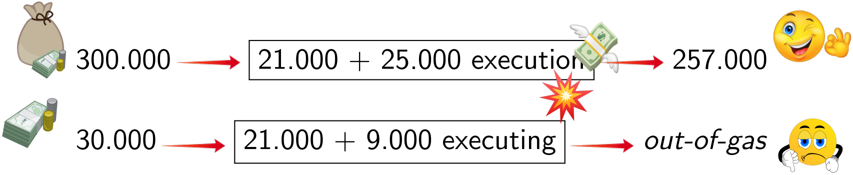
- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)

300.000 → $\boxed{\text{21.000 + 25.000 execution}}$  → 257.000 

30.000 → $\boxed{\text{21.000 + 9.000 executing}}$

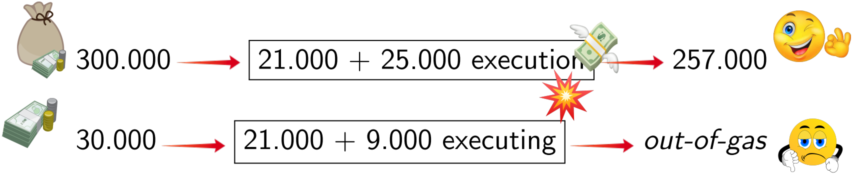
Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)



Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction.
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- EVM specification provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 22.100)



- Rationale of gas metering :
  - prevents attacks based on non-terminating executions;
  - avoids wasting *miners* computational resources;
  - discourages users to overuse replicated *storage*

what is Gastap?

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];
      }}
  }
  ...
}
```

WHAT IS GASTAP?

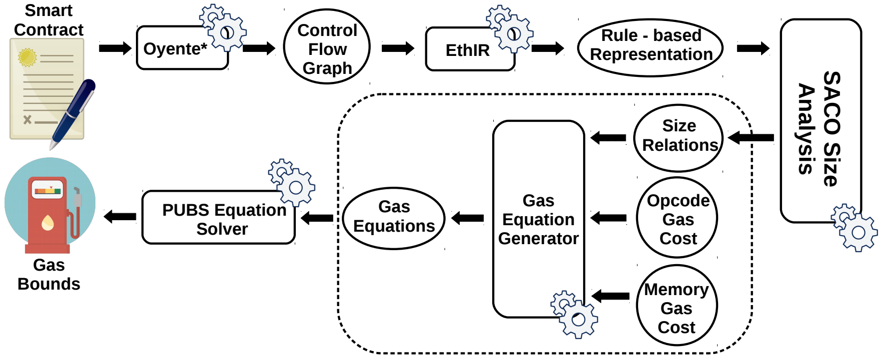
GASTAP

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];
      }}
  }
  ...
}
```

15+ (1555+779·slots)

Arquitecture of Gastap



what is EthIR?

EthIR

EthIR is a framework that translates Ethereum bytecode into an intermediate representation.

Ethereum Bytecode

- Stack based language.
- Small set of instructions.

```
PUSH 0X01
PUSH 0X04
ADD
PUSH 0X000F
JUMP
```

EthIR (decompiler from EVM)

- Why do we work at bytecode level?

EthIR (decompiler from EVM)

- Why do we work at bytecode level?
  - The source code of the smart contracts is not available
    - Blockchain only stores the bytecode

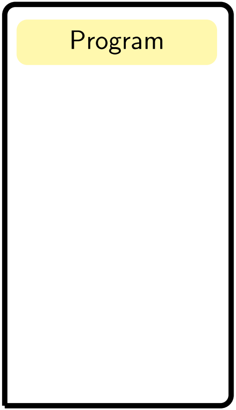
EthIR (decompiler from EVM)

- Why do we work at bytecode level?
  - The source code of the smart contracts is not available
    - Blockchain only stores the bytecode
  - Information only available at bytecode level
    - Gas consumption

EthIR (decompiler from EVM)

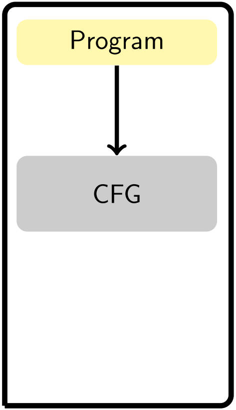
- Why do we work at bytecode level?
  - The source code of the smart contracts is not available
    - Blockchain only stores the bytecode
  - Information only available at bytecode level
    - Gas consumption
  - Analyses may be affected by compiler optimizations
    - Binary arithmetic, unfolding, etc.

How does EthIR work?



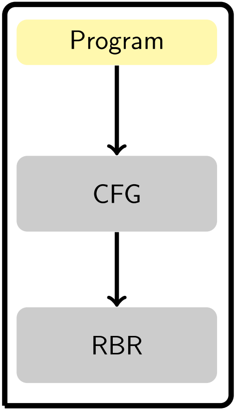
- Program: Solidity, EVM Bytecode or Disassembly code.

How does EthIR work?



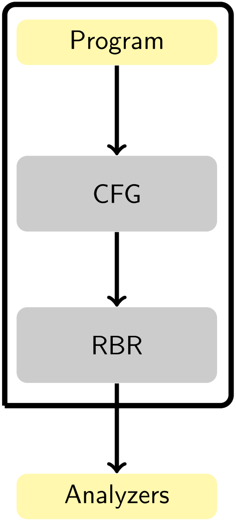
- Program: Solidity, EVM Bytecode or Disassembly code.
- CFG: Detect jump addresses

How does EthIR work?



- Program: Solidity, EVM Bytecode or Disassembly code.
- CFG: Detect jump addresses
- RBR: Rule based Representation.
  Make explicit the stack, memory, fields variables.

How does EthIR work?



- Program: Solidity, EVM Bytecode or Disassembly code.
- CFG: Detect jump addresses
- RBR: Rule based Representation. Make explicit the stack, memory, fields variables.
- Analyzers: The RBR can be injected straightforward to existing analyzers with minor changes.

How does EthIR work?

```
contract Loop{

  uint sum = 0;
  uint number = 5;

  function multiply(uint a){
    for(uint i = 0; i<a; i++){
      sum = sum+number;
    }
  }
```

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

- We generate the CFG of the smart contract including annotations
  - SSTORE, SSLOAD, MSTORE, MLOAD, CALLDATALOAD...

How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 1
    s(4) = g(1)
    s(5) = 0
    s(5) = g(0)
    s(4) = s(5)+s(4)
    s(5) = 0
    s(6) = s(4)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)
    g(0) = s(5)
    ...
    s(4) = 179
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
```

How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 1
    s(4) = g(1)
    s(5) = 0
    s(5) = g(0)
    s(4) = s(5)+s(4)
    s(5) = 0
    s(6) = s(4)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)
    g(0) = s(5)
    ...
    s(4) = 179
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
```

- Each block corresponds to one rule.

How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 1
    s(4) = g(1)
    s(5) = 0
    s(5) = g(0)
    s(4) = s(5)+s(4)
    s(5) = 0
    s(6) = s(4)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)
    g(0) = s(5)
    ...
    s(4) = 179
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
```

- Each block corresponds to one rule.
- The fields and data are passed to the rules as parameters.

How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 1
    s(4) = g(1)
    s(5) = 0
    s(5) = g(0)
    s(4) = s(5)+s(4)
    s(5) = 0
    s(6) = s(4)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)
    g(0) = s(5)
    ...
    s(4) = 179
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
```

- Each block corresponds to one rule.
- The fields and data are passed to the rules as parameters.
- Stack variables are explicit in the code.

How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 1
    s(4) = g(1)
    s(5) = 0
    s(5) = g(0)
    s(4) = s(5)+s(4)
    s(5) = 0
    s(6) = s(4)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)
    g(0) = s(5)
    ...
    s(4) = 179
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
```

- Each block corresponds to one rule.
- The fields and data are passed to the rules as parameters.
- Stack variables are explicit in the code.
- Jump instructions are converted to calls to new blocks.

How does EthIR work?

```
start address: 175
end address: 170
type: cond
jump target: 189
falls to: 171

JUMPDEST
PUSH1 0x05
PUSH1 0x01
DUP2
SWAP1
SSTORE 1
PUSH1 0xab
LT
PUSH1 0xbd
JUMPI
```

```
block175(s(3), s(2), s(1), s(0), g(1), g(0)) ⇒
    s(4) = 5
    s(5) = 1
    s(6) = s(4)
    s(7) = s(6)
    s(6) = s(5)
    s(5) = s(7)
    g(1) = s(5)
    s(4) = 171
    call(jump175(s(4),s(3),s(2),s(1),s(0),g(1),g(0)))

jump175(s(4),s(3),s(2),s(1),s(0),g(1),g(0)) ⇒
    lt(s(4),s(3))
    call(block189(s(2),s(1),s(0),g(1)))

jump179(s(4),s(3),s(2),s(1),s(0),g(1),g(0)) ⇒
    geq(s(4),s(3))
    call(block171(s(2),s(1),s(0),g(1),g(0)))
```

- Each block corresponds to one rule.
- The fields and data are passed to the rules as parameters.
- Stack variables are explicit in the code.
- Jump instructions are converted to calls to new blocks.

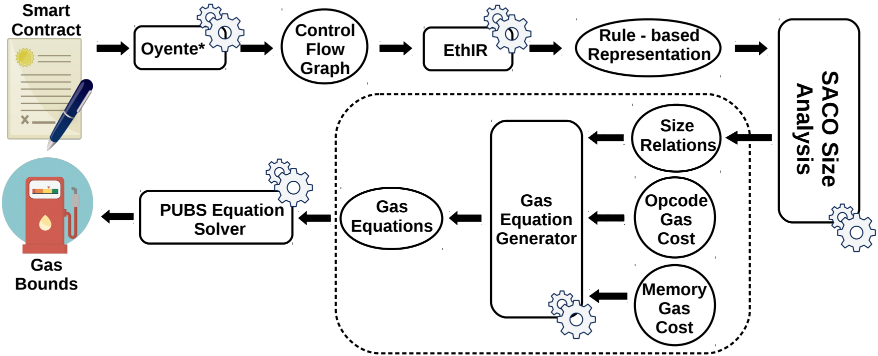
How does EthIR work?

```
start address: 187
end address: 210
type: uncond
jump target: 179
PUSH1 0x01
SLOAD 1
PUSH1 0x00
SLOAD 0
ADD
PUSH1 0x00
DUP2
SWAP1
SSTORE 0
...
PUSH1 0xb3
JUMP
```

```
block187(s(3), s(2), s(1), s(0), g(1), g(0))  ⇒
    s(4) = 1   nop(PUSH1)
    s(4) = g(1)   nop(SLOAD)
    s(5) = 0   nop(PUSH1)
    s(5) = g(0)   nop(SLOAD)
    s(4) = s(5)+s(4)   nop(ADD)
    s(5) = 0   nop(PUSH1)
    s(6) = s(4)   nop(DUP2)
    s(7) = s(5)
    s(5) = s(6)
    s(6) = s(7)   nop(SWAP1)
    g(0) = s(5)   nop(SSTORE)
    ...
    s(4) = 179   nop(PUSH1)
    call(block179(s(3),s(2),s(1),s(0),g(1), g(0)))
    nop(JUMP)
```

- Each block corresponds to one rule.
- The fields and data are passed to the rules as parameters.
- Stack variables are explicit in the code.
- Jump instructions are converted to calls to new blocks.
- **nop** instructions for the cost model.

Arquitecture of Gastap



Gas Equations

- Generated using the RBR and SR
- It depends on the opcodes executed and the memory used during the transaction
  - Memory Gas.
  - Opcode Gas.

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

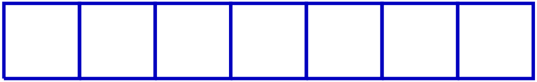
- Gas consumed by EVM instruction i.
- μ'_i and μ_i are the highest memory slot accessed after and before the execution of the opcode at program point i, resp.

Gas Equations

Memory Gas

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

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

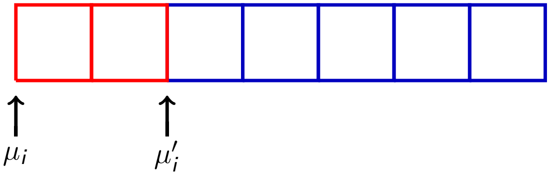


Gas Equations

Memory Gas

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

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

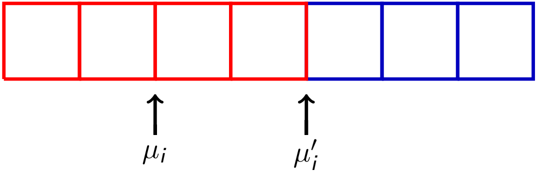


Gas Equations

Memory Gas

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

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

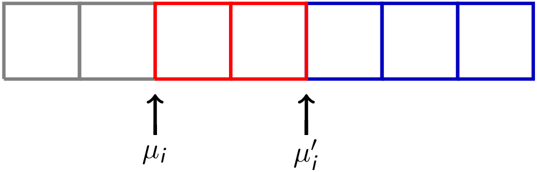


Gas Equations

Memory Gas

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

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



Gas Equations

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

Gas Equations

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(8)*, ADD(3), MLOAD(3), etc.
  - Different constant gas depending on some condition: SLOAD(100 / 2100)*, SSTORE (100 to 22100)*, etc.
  - Non-constant gas consumption: EXP, SHA3, etc.
  - * These fees changed after Berlin hard fork.

Gas Equations

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), CREATE(32000),...

Gas Equations

Opcode Gas

- **Constant gas depending on some condition**

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

When loading some value form address *pos*:

$$\textbf{SLOAD} = \begin{cases} 100 & \text{if} \quad \text{storage(pos) previously accessed (in the current or a} \\ 2100 & \text{otherwise} \end{cases}$$

SSTORE has a more complex constant gas expression that depends on several conditions (see Yellow Paper):

- The location has been accessed in a previous transaction,
- it has been already modified in this transaction,
- and the value being assigned to that storage location.

Gas Equations

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 Equations

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

- Each rule is transformed into a cost equation
- Nop instructions determine the gas consumed
- Calls to other rules replaced to cost equations
- Size relations attached
- PUBS Solver: Generation of a closed-form gas upper bound

```
block1619(s(4),··· , s(0), g(3), g(1), l(l2), l(l1), l(l0))⇒
    s(5) = 3    nop(PUSH1)
    s(6) = s(5)   nop(DUP1)
    s(6) = g(3)   nop(SLOAD)
    ···
    s(6) = s(4)   nop(DUP2)
       call(jump1619(s(6),··· ,s(0),g(3),  g(1),l(l2),  l(l1),
l(l0)))
   nop(LT)  nop(ISZERO)  nop(PUSH2)  nop(JUMPI)
```

```
block1619(s(4),··· , s(0), g(3), g(1), l(l2),··· )⇒
  234+jump1619(s(6),··· ,s(0),g(3), g(1),··· )
  {s(6) = s(4), s(5) = g(3)}
```

Gas Equations

- We compute the memory gas cost and opcode gas cost separately
  - Memory gas cost: Infer the highest slot of memory accesed by the opcodes executed in the function
  - Opcode gas: Compute gas equations

Total gas consumption = Memory gas + Opcode gas

Practice with Gastap

Gastap can be found at `https://costa.fdi.ucm.es/gastap/`

1. Load into Gastap the following contract and analyze it.
    - Copy the code on Gastap central panel and press **Refresh Outline**.
    - Then Select one function on the right and press **Apply**.
      Analyze the contract functions one by one.
2. Explain the results obtained for these functions.

```
pragma solidity ^0.4.0;
contract exercise3 {
  uint [] arr = new uint[](5);
  function p1() external view returns (uint) {
    uint sumEven = 0;
    for (uint i = 0; i < arr.length; i+=2) {
      sumEven += arr[i];
    }
    return sumEven;
  }
  function p2() external view { uint [] memory local = arr; }
  function p3() external view { uint [] storage local = arr; }
}
```

GASOL: Gas Analysis Toolkit

Goal: Optimization of gas consumption. It works at two levels:

1. Solidity level: It uses resource analysis to reduce the accesses to storage.
2. EVM bytecode level: It uses superoptimization to find equivalent bytecode that produces the same state but consuming less gas.

GASOL: Solidity Level (Resource Analysis)

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 = totalSupply + amount;
   }
 }
```

GASOL: Solidity Level (Resource Analysis)

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 = totalSupply + amount;
   }
 }
```

GASOL: Solidity Level (Resource Analysis)

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 = totalSupply + amount;
   }
 }
```

Accesses to `totalSupply`

2*data

GASOL: Solidity Level (Resource Analysis)

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 = totalSupply + amount;
   }
 }
```

Accesses to `totalSupply`

2*data

Sound transformation?

GASOL: Solidity Level (Resource Analysis)

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 = totalSupply + amount;
   }
 }
```

Accesses to totalSupply

2*data

Optimized program:

```
uint totalSupply_mem = 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_mem = totalSupply_mem + amount;
   }
}
totalSupply = totalSupply_mem;
```

Sound transformation? **Yes**

GASOL: Solidity Level (Resource Analysis)

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

GASOL: EVM Level (Superoptimization)

Superoptimization

Superoptimization is a transformation technique that aims to find the (cost-optimal) translation of a code by trying all posible sequences of instructions that produce the same result.

GASOL: EVM Level (Superoptimization)

Superoptimization

Superoptimization is a transformation technique that aims to find the (cost-optimal) translation of a code by trying all posible sequences of instructions that produce the same result.

- **given:** source program **s** and a cost function **C**
- **find:** target program **t** that
  1. has **minimal** cost **C(t)**
  2. correctly implements s
- **using:** constraint solver

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

GASOL: GAS OPTIMIZATION TOOLKIT

ethereum foundation

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

```
SWAP1
PUSH1 0
MSTORE
PUSH1 32
MSTORE
ISZERO
```

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

SWAP1
PUSH1 0
MSTORE
PUSH1 32
MSTORE
ISZERO

18 gas

8 bytes

GASOL: GAS Optimization tooLkit

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

SWAP1
PUSH1 0
MSTORE
PUSH1 32
MSTORE
ISZERO

18 gas

8 bytes

Stack	ISZERO(s2)
Memory	MEM[0 $\mapsto$ s1, 32 $\mapsto$ s0]
Storage	

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

SWAP1
PUSH1 0
MSTORE
PUSH1 32
MSTORE
ISZERO

18 gas

8 bytes

Stack	ISZERO(s2)
Memory	MEM[0 $\mapsto$ s1, 32 $\mapsto$ s0]
Storage	

PUSH1 32
MSTORE
PUSH1 0
MSTORE
ISZERO

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and bytes-size superoptimization tool that uses symbolic execution, dependency analysis and a Max-SMT solver for stack/memory/storage optimization

```
SWAP1
PUSH1 0
MSTORE
PUSH1 32
MSTORE
ISZERO
```

18 gas

8 bytes

15 gas

7 bytes

```
PUSH1 32
MSTORE
PUSH1 0
MSTORE
ISZERO
```

Stack	ISZERO(s2)
Memory	MEM[0 $\mapsto$ s1, 32 $\mapsto$ s0]
Storage	

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
PUSH1 96      SHL
SWAP2         SUB
SWAP1         NOT
SWAP2         AND
SHL           PUSH1 128
PUSH1 1       MSTORE
PUSH1 1       PUSH1 160
PUSH1 96      MSTORE
```

Stack

s0
s1

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

→ PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
SHL	PUSH1 128
PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

96
s0
s1

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
  PUSH1 96     SHL
→ SWAP2        SUB
  SWAP1        NOT
  SWAP2        AND
  SHL          PUSH1 128
  PUSH1 1      MSTORE
  PUSH1 1      PUSH1 160
  PUSH1 96     MSTORE
```

Stack

s1
s0
96

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
  PUSH1 96     SHL
  SWAP2        SUB
→ SWAP1        NOT
  SWAP2        AND
  SHL          PUSH1 128
  PUSH1 1      MSTORE
  PUSH1 1      PUSH1 160
  PUSH1 96     MSTORE
```

Stack

s0
s1
96

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

	PUSH1 96	SHL
	SWAP2	SUB
	SWAP1	NOT
→	SWAP2	AND
	SHL	PUSH1 128
	PUSH1 1	MSTORE
	PUSH1 1	PUSH1 160
	PUSH1 96	MSTORE

Stack

96
s1
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
→ SHL	PUSH1 128
PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
SHL	PUSH1 128
→ PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

1
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
  PUSH1 96
  SWAP2
  SWAP1
  SWAP2
  SHL
  PUSH1 1
→ PUSH1 1
  PUSH1 96
  SHL
  SUB
  NOT
  AND
  PUSH1 128
  MSTORE
  PUSH1 160
  MSTORE
```

Stack

1
1
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
  PUSH1 96     SHL
  SWAP2        SUB
  SWAP1        NOT
  SWAP2        AND
  SHL          PUSH1 128
  PUSH1 1      MSTORE
  PUSH1 1      PUSH1 160
→ PUSH1 96     MSTORE
```

Stack

96
1
1
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	→ SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
SHL	PUSH1 128
PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

SHL(96, 1)
1
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
PUSH1 96        SHL
SWAP2        →  SUB
SWAP1           NOT
SWAP2           AND
SHL             PUSH1 128
PUSH1 1         MSTORE
PUSH1 1         PUSH1 160
PUSH1 96        MSTORE
```

Stack

SUB(SHL(96, 1), 1)
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
PUSH1 96
SWAP2
SWAP1
SWAP2
SHL
PUSH1 1
PUSH1 1
PUSH1 96
SHL
SUB
→ NOT
AND
PUSH1 128
MSTORE
PUSH1 160
MSTORE
```

Stack

NOT(SUB(SHL(96, 1), 1))
SHL(96, s1)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	→ AND
SHL	PUSH1 128
PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

AND(NOT(SUB(SHL(96, 1), 1)), ...)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
SHL	→ PUSH1 128
PUSH1 1	MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

128
AND(NOT(SUB(SHL(96, 1), 1)), ...)
s0

Memory

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

PUSH1 96	SHL
SWAP2	SUB
SWAP1	NOT
SWAP2	AND
SHL	PUSH1 128
PUSH1 1	→ MSTORE
PUSH1 1	PUSH1 160
PUSH1 96	MSTORE

Stack

s0

Memory

MSt(128, AND(NOT(SUB(SHL(96,1),1)), ...))

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
PUSH1 96          SHL
SWAP2             SUB
SWAP1             NOT
SWAP2             AND
SHL               PUSH1 128
PUSH1 1           MSTORE
PUSH1 1        →  PUSH1 160
PUSH1 96          MSTORE
```

Stack

160
s0

Memory

MSt(128, AND(NOT(SUB(SHL(96,1),1)), ...))

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack

```
PUSH1 96      SHL
SWAP2         SUB
SWAP1         NOT
SWAP2         AND
SHL           PUSH1 128
PUSH1 1       MSTORE
PUSH1 1       PUSH1 160
PUSH1 96   →  MSTORE
```

Stack

Memory

MSt(128, AND(NOT(SUB(SHL(96,1),1)), ...))
MSt(160, s0)

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack
- Simplification rules based on semantics of bytecodes: integers, units, idempotence...

```
PUSH1 96    SHL
SWAP2       SUB
SWAP1       NOT
SWAP2       AND
SHL         PUSH1 128
PUSH1 1     MSTORE
PUSH1 1     PUSH1 160
PUSH1 96    MSTORE
```

Stack

Memory

MSt(128, AND(NOT(SUB(SHL(96,1),1)), ...))
MSt(160, s0)

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack
- Simplification rules based on semantics of bytecodes: integers, units, idempotence...

```
PUSH1 96     SHL
SWAP2        SUB
SWAP1        NOT
SWAP2        AND
SHL          PUSH1 128
PUSH1 1      MSTORE
PUSH1 1      PUSH1 160
PUSH1 96     MSTORE
```

Stack

Memory

MSt(128, AND(NOT(SUB(SHL(96,1),1)), ...))
MSt(160, s0)

Analysis 1: Symbolic Execution

- Functional description of the final Stack and Memory and Storage (SMS) in terms of the initial stack after executing the instructions in each block of the CFG
- Obtained by symbolic execution from the initial stack
- Simplification rules based on semantics of bytecodes: integers, units, idempotence...

```
PUSH1 96      SHL
SWAP2         SUB
SWAP1         NOT
SWAP2         AND
SHL           PUSH1 128
PUSH1 1       MSTORE
PUSH1 1       PUSH1 160
PUSH1 96      MSTORE
```

Stack

Memory

MSt(128, AND(1.158×10^{29}, ...))
MSt(160, s0)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization

→ PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE

...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
ISZERO

Stack

Memory

Storage

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization

PUSH1 64 DUP1 ADD PUSH1 64 MSTORE ... SLOAD ... SSTORE	... SLOAD ... SSTORE ... CALLVALUE DUP1 → ISZERO

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

```
PUSH1 64        ...
DUP1            SLOAD
ADD             ...
PUSH1 64        SSTORE
MSTORE          ...
...             CALLVALUE
SLOAD           DUP1
...           → ISZERO
SSTORE
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(**1**)	SSTORE(**1**,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(**1**)	SSTORE(1,V1)	SLOAD(1)	SSTORE(**1**,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
  ...
  SLOAD
  ...
  SSTORE
  ...
  CALLVALUE
  DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies

PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE

...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies
- Apply simplification rules on memory

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies
- Apply simplification rules on memory

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)	SLOAD(1)	SSTORE(1,V2)

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies
- Apply simplification rules on memory

```
PUSH1 64          ...
DUP1              SLOAD
ADD               ...
PUSH1 64          SSTORE
MSTORE            ...
...               CALLVALUE
SLOAD             DUP1
...             → ISZERO
SSTORE
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

V2' = V2[SLOAD(1)↦V1]

SLOAD(1)	SSTORE(1,V1)		SSTORE(1,V2')

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies
- Apply simplification rules on memory

```
PUSH1 64
DUP1
ADD
PUSH1 64
MSTORE
...
SLOAD
...
SSTORE
```

```
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
→ ISZERO
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)	SSTORE(1,V1)		SSTORE(1,V2')

Analysis 2: dependency analysis

- Static analysis applied to memory for more optimization
- Compute dependencies
- Apply simplification rules on memory

```
PUSH1 64        ...
DUP1            SLOAD
ADD             ...
PUSH1 64        SSTORE
MSTORE          ...
...             CALLVALUE
SLOAD           DUP1
...           → ISZERO
SSTORE
```

Stack

ISZ(CALLVALUE)
CALLVALUE

Memory

MSTORE(64,128)

Storage

SLOAD(1)			SSTORE(1,V2')

Notion of preorder

- After the simplifications, we use the **dependencies** to define a pre-order $\sqsubset$ *(happens-before)* among memory accesses the SMT solver must respect when finding optimal sequences

Notion of preorder

- After the simplifications, we use the **dependencies** to define a pre-order $\sqsubset$ *(happens-before)* among memory accesses the SMT solver must respect when finding optimal sequences
  - In the previous example, SLOAD(1) $\sqsubset$ SSTORE($1, V2'$) and MSTORE(64,128) is not related to any instruction.

Notion of preorder

- After the simplifications, we use the **dependencies** to define a pre-order $\sqsubset$ *(happens-before)* among memory accesses the SMT solver must respect when finding optimal sequences
  - In the previous example, SLOAD(1) $\sqsubset$ SSTORE($1, V2'$) and MSTORE(64,128) is not related to any instruction.
- Extension to other uninterpreted functions: if $B(\ldots, A(\ldots), \ldots)$ then $A \sqsubset B$

Notion of preorder

- After the simplifications, we use the **dependencies** to define a pre-order $\sqsubset$ *(happens-before)* among memory accesses the SMT solver must respect when finding optimal sequences
  - In the previous example, SLOAD(1) $\sqsubset$ SSTORE$(1, V2')$ and MSTORE(64,128) is not related to any instruction.
- Extension to other uninterpreted functions: if $B(\ldots, A(\ldots), \ldots)$ then $A \sqsubset B$
  - In previous example, CALLVALUE $\sqsubset$ ISZ(CALLVALUE)

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver
    1. Non-stack operations considered as uninterpreted functions

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver
  1. Non-stack operations considered as uninterpreted functions
  2. Abstract all the non-store subexpressions that appear in the SMS

$s2 = \mathrm{SHL}(96, s1)$

$s3 = \mathrm{AND}(1.158 \times 10^{29}, s2)$

Stack

Memory

MSTORE(128, ...)
MSTORE(160, s0)

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver
  1. Non-stack operations considered as uninterpreted functions
  2. Abstract all the non-store subexpressions that appear in the SMS

$s2 = \mathrm{SHL}(96, s1)$ $s3 = \mathrm{AND}(1.158 \times 10^{29}, s2)$

Stack

Memory

MSTORE(128, ...)
MSTORE(160, s0)

$\longrightarrow$

Stack

Memory

MSTORE(128, s3)
MSTORE(160, s0)

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver
  1. Non-stack operations considered as uninterpreted functions
  2. Abstract all the non-store subexpressions that appear in the SMS
  3. Bounds on the number of opcodes and on the size of the stack

$\mathrm{s2} = \mathrm{SHL}(96, \mathrm{s1})$ $\qquad$ $\mathrm{s3} = \mathrm{AND}(1.158 \times 10^{29}, \mathrm{s2})$

Stack		**Stack**
Memory	$\longrightarrow$	**Memory**
MSTORE(128, ...)		MSTORE(128, s3)
MSTORE(160, s0)		MSTORE(160, s0)

Analysis 3: Optimal Synthesis using Max-SMT

- Balance between a great coverage of EVM optimization and an encoding in a simple theory for a SMT solver
  1. Non-stack operations considered as uninterpreted functions
  2. Abstract all the non-store subexpressions that appear in the SMS
  3. Bounds on the number of opcodes and on the size of the stack

$s2 = SHL(96, s1)$ $s3 = AND(1.158 \times 10^{29}, s2)$

$maxLn = 10$ $skSz = 5$

Stack

Memory
MSTORE(128, ...)
MSTORE(160, s0)

$\longrightarrow$

Stack

Memory
MSTORE(128, s3)
MSTORE(160, s0)

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

$\mathrm{t}_j = \mathrm{DUP}k \rightarrow$

$\neg u_{skSz-1,j} \wedge u_{k-1,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k-1,j} \wedge \mathit{MoveRest}$

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula $\Rightarrow$ hard constraints

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize:

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize:
    - Gas model

Analysis 3: Optimal Synthesis using Max-SMT

- Complete encoding
  1. Constraint to describe the initial stack
  2. Constraint to describe the target stack
  3. Constraint to model the stack variables
  4. Constraint to describe effect of instructions on the stack
  5. Constraint to preserve the order among memory accesses

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize:
    - Gas model
    - Bytes-size model

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

$\longrightarrow$

PUSH1 160

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

$\longrightarrow$

PUSH1 160
MSTORE

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

(objectives (gas 24))

((t_0 0)) ((a_3 160))
((t_1 13)) ((_))
((t_2 0)) ((a_2 96))
((t_3 14)) ((_))
((t_4 0)) ((a_3 1.158×10^{29}))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

...

((t_5 11)) ((_))
((t_6 0)) ((a_6 128))
((t_7 12)) ((_))
((t_8 2)) ((_))
((t_9 2)) ((_))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

...

((t_5 11)) ((_))
((t_6 0)) ((a_6 128))
((t_7 12)) ((_))
((t_8 2)) ((_))
((t_9 2)) ((_))

→

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND
PUSH1 128

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

```
sat

...

( (t5 11) ) ( (_) )
( (t6 0) ) ( (a6 128) )
( (t7 12) ) ( (_) )
( (t8 2) ) ( (_) )
( (t9 2) ) ( (_) )
```

→

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND
PUSH1 128
MSTORE

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:

sat

...

((t_5 11)) ((_))
((t_6 0)) ((a_6 128))
((t_7 12)) ((_))
((t_8 2)) ((_))
((t_9 2)) ((_))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND
PUSH1 128
MSTORE

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:
- Initial block consumes 48 gas and 22 bytes

sat

...

((t_5 11)) ((_))
((t_6 0)) ((a_6 128))
((t_7 12)) ((_))
((t_8 2)) ((_))
((t_9 2)) ((_))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND
PUSH1 128
MSTORE

Optimized Bytecode from Model

- The optimized bytecode is obtained from the model together with the cost:
- Initial block consumes 48 gas and 22 bytes
- Final block consumes 24 gas and 43 bytes

sat

...

((t_5 11)) ((_))
((t_6 0)) ((a_6 128))
((t_7 12)) ((_))
((t_8 2)) ((_))
((t_9 2)) ((_))

$\longrightarrow$

PUSH1 160
MSTORE
PUSH1 96
SHL
PUSH32 1.158×10^{29}
AND
PUSH1 128
MSTORE

Conclusions

- Analysis and optimization of Ethereum smart contracts is a popular research topic
  - There are tools based on symbolic execution, SMT solving or certified programming for detecting security vulnerabilities
- Presented tools:
  - GASTAP
    - estimate the gas fee for running transactions
    - attackers: estimate how much *Ether* an adversary has to pour into a contract in order to execute an out-of-gas attack
    - analyzer that measures several resources
  - GASOL
    - optimize storage accesses at Solidity level
    - optimize the gas fee for running transactions and bytes-size
    - Superoptimization and Max-SMT solver
  - tools applied to thousands of real smart contracts