Program Analysis – 184.703 – 2022W

Precise static modeling of Ethereum “memory”

Sifis Lagouvardos, Neville Grech, Ilias Tsatiris,
Yannis Smaragdakis:
Precise static modeling of Ethereum "memory".
Proc. ACM Program. Lang. 4(OOPSLA): 190:1-
190:26 (2020)

Simão Costa (e12202234)

Ethereum: a programmable blockchain

- Decentralized register of transactions that also serves as the execution environment for **smart contracts**:

Programs stored on a blockchain that run when predetermined conditions are met

Written mainly in a Turing-complete programming language - **Solidity**
The code is first compiled to bytecode and then is deployed to the blockchain.

Smart contracts are mostly only available as byte code and not as source code!

- A **gas cost** is paid for performing transactions.

Proportional to the complexity of the computation

Ethereum: a programmable blockchain

An Ethereum user (sender) can submit a transaction proposal to the Ethereum network, containing:

(i) sender and receiver fields

(ii) a message

(iii) value

(iv) a gas budget

i	sender: 0xBEEFBABE; receiver: 0xC0CAC01A
ii	deliverCans(amount: 10, to: 0xCAFED00D)
iii	2 ETH (~ $400)
iv	10000 gas units @ 2 ∗ 10−7ETH each

EVM - Ethereum Virtual Machine

- Abstract machine description on which smart contracts are executed by EVM emulators.
- Computes the state of the Ethereum network after a new block is added to the chain.
- Stack based
- The EVM memory model distinguishes **storage** and **memory**:

  Storage: a persistent data store, kept on the blockchain's state

  Memory: used as a temporary store for all sorts of data (zero-set when any transaction is started)

Smart contract example

```
contract Example{
   string onStorage;

   function setIt(string memory newStr) public {
       onStorage = newStr;
   }

   function getHash() public view returns (bytes32){
       return keccak256(onStorage);
   }
}
```

Variable stored on persistent storage

Hashes memory contents

“Memory” hides implicit computation!

```
mapping ( string => string ) mTokens ; ...
function getToken ( string pDocumentHash ) view public returns ( string )
{ return mTokens [ pDocumentHash ]; }
```

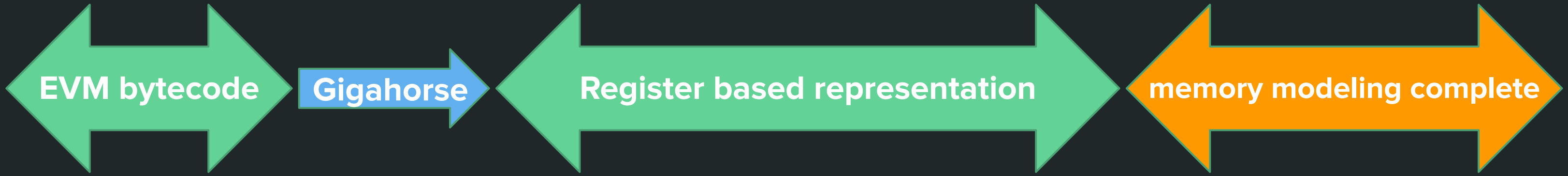
```
function getToken(var arg0) returns (var r0) {
  var var0 = 0x053b;  var0 = func_06C6();  var var1 = 0x02;
  var temp0 = arg0;  var var2 = temp0;
  var var3 = memory[0x40:0x60];  var var4 = var3;
  var var5 = var2 + 0x20;  var var6 = memory[var2:var2 + 0x20];
  var var7 = var6;  var var8 = var4;  var var9 = var5;
  if (var7 < 0x20) {
  label_0573:
    var temp1 = 0x0100 ** (0x20 - var7) - 0x01;  var temp2 = var8;
    memory[temp2:temp2 + 0x20] = (memory[var9:var9 + 0x20] & ~temp1) | (memory[temp2:temp2 + 0x20] & temp1);
    var temp3 = var6 + var4;
    memory[temp3:temp3 + 0x20] = var1;
    var temp4 = memory[0x40:0x60];
    var temp5 = keccak256(memory[temp4:temp4+(temp3+0x20)-temp4]);
    var temp6 = storage[temp5];
    var temp7 = (!(temp6 & 0x01) * 0x0100 - 0x01 & temp6) / 0x02;
    var temp8 = memory[0x40:0x60];
    memory[0x40:0x60] = temp8 + (temp7+0x1f) / 0x20 * 0x20 + 0x20;
    var1 = temp8;  var2 = temp5;  var3 = temp7;
    memory[var1:var1 + 0x20] = var3;
    var4 = var1 + 0x20;  var5 = var2;
    var temp9 = storage[var5];
    var6 = (!(temp9 & 0x01) * 0x0100 - 0x01 & temp9) / 0x02;
    if (!var6) {
    label_063A:
      return var1;
    } else if (0x1f < var6) {
      var temp10=var4; var temp11 = temp10 + var6; var4=temp11;
      memory[0x00:0x20] = var5;
      var temp12 = keccak256(memory[0x00:0x20]);
      memory[temp10:temp10 + 0x20] = storage[temp12];
      var5 = temp12 + 0x01;  var6 = temp10 + 0x20;
      if (var4 <= var6) { goto label_0631; }
    label_061D:
      var temp13 = var5;  var temp14 = var6;
      memory[temp14:temp14 + 0x20] = storage[temp13];
      var5 = temp13 + 0x01;  var6 = temp14 + 0x20;
      if (var4 > var6) { goto label_061D; }
    label_0631:
      var temp15 = var4; var temp16 = temp15+(var6 - temp15&0x1f);
      var6 = temp15;  var4 = temp16;
      goto label_063A;
    } else {
      var temp17 = var4;
      memory[temp17:temp17+0x20] = storage[var5]/0x0100 * 0x0100;
      var4 = temp17 + 0x20;  var6 = var6;
      goto label_063A;
    }
  } else {
  label_0559:
    var temp18 = var9;  var temp19 = var8;
    memory[temp19:temp19 + 0x20] = memory[temp18:temp18 + 0x20];
    var8 = temp19 + 0x20; var9 = temp18 + 0x20; var7 = var7-0x20;
    if (var7 < 0x20) { goto label_0573; }
    else { goto label_0559; }
  }
}
```

Smart contract example (decompiled)

0x116: PUSH1 0x1
(...)
0x1c5: PUSH10x0
0x1c7: MSTORE
0x1c8: PUSH10x20
0x1ca: PUSH1 0x0
0x1cc: SHA3
(...)

0x116: v116 = 0x1
(...)
0x1c5: v1c5 = 0x0
0x1c7: MSTORE v1c5(0x0) = v116(0x1)
0x1c8: v1c8 = 0x20
0x1ca: v1ca = 0x0
0x1cc: v1cc = SHA3 v1ca(0x0) v1c8(0x20)
(...)

0x116: v116 = 0x1
(...)
0x1cc: v1cc = SHA3 [v116(0x1)]
(...)



Precise Memory Modeling

The analysis tracks symbolic values based on the **free-memory pointer**, offset by a constant.

maintained at a constant memory address (0x40/decimal 64)

Use constant and symbolic indexes to infer high-level properties:

- **Arguments** passed through memory, to statements that read from it
- **Array** allocations, reads, writes
- Access to the **data returned** by external calls

Analysis - Input

- The analysis is expressed as a set of declarative rules, using datalog.

<u>Concrete instructions</u>

(s: S) :[*r* : *V* := ADD/SUB/MUL(*a*: *V* ∪ *C*, *b*: *V* ∪ *C*)]

(s: S) :[MSTORE(*addr* : *V*, *from* : *V*)]

(s: S) :[*to* : *V* := MLOAD(*addr* : *V*)]

Arithmetic operations

Memory related operations

<u>Generic instructions / Syntactic patterns</u>

STATEMENTUSESMEMORY(*s* : *S*, *start* : *V*, *len* : *V*)

Memory consuming operations

V is a set of program variables
C is a set of constants, C ⊆ N256

S is a set of statement identifiers
N256 is the set of 256-bit unsigned integers

M is a set of symbolic values
FreePtr is the free-memory pointer

Analysis - Interfacing with other analysis modules

Variable_Value(v : V, c : C)
Flows(from : V, to : V)
Alias(x : V, y : V)

Constant propagation/folding

MatchingMSTORE(ms : S, s : S)

MSTORE statement 'ms' writes for memory

UnchangedFreePointer(s1 : S, s2 : S)

Analysis - Symbolic Value creation

```
Variable_Value+(to, val),
FreePointerBasedValue(val, mload, 0) :-
          mload: [to := MLOAD(FreePtr)], val = mload ++ "0x0"

Variable_Value+(to, val),
FreePointerBasedValue(val, mload, res) :-
 [to := ADD(numVar, freePtrBasedVar)],
 Variable_Value(numVar, numVal1),
 Variable_Value+(freePtrBasedVar, freePtrBasedVal),
 FreePointerBasedValue(freePtrBasedVal, mload, numVal2),
 res = numVal1 + numVal2,
 val = mload ++ numVal1 + numVal2.
```

Symbolic value creation on MLOADs on the free-memory pointer

Constant folding of the numeric part of the symbolic values

Analysis - Sample of final outputs

```
StatementUsesMemoryAtIndex(stmt, relativeIndex, actual) :-
  StatementUsesMemory(stmt, startVar, lenVar),
  Variable_Value+(startVar, startVal),
  Variable_Value(lenVar, lenVal),
  MatchingMSTORE(mstore, stmt),
  mstore: [MSTORE(indexVar, actual)],
  Variable_Value+(indexVar, indexVal),
  FreePointerDiff(indexVal, startVal, relativeIndex),
  lenVal > relativeIndex >= 0

StatementUsesMemory_ActualMemoryArg(stmt, i, actual) :-
  order(StatementUsesMemoryAtIndex(stmt, _, actual)) = i
```

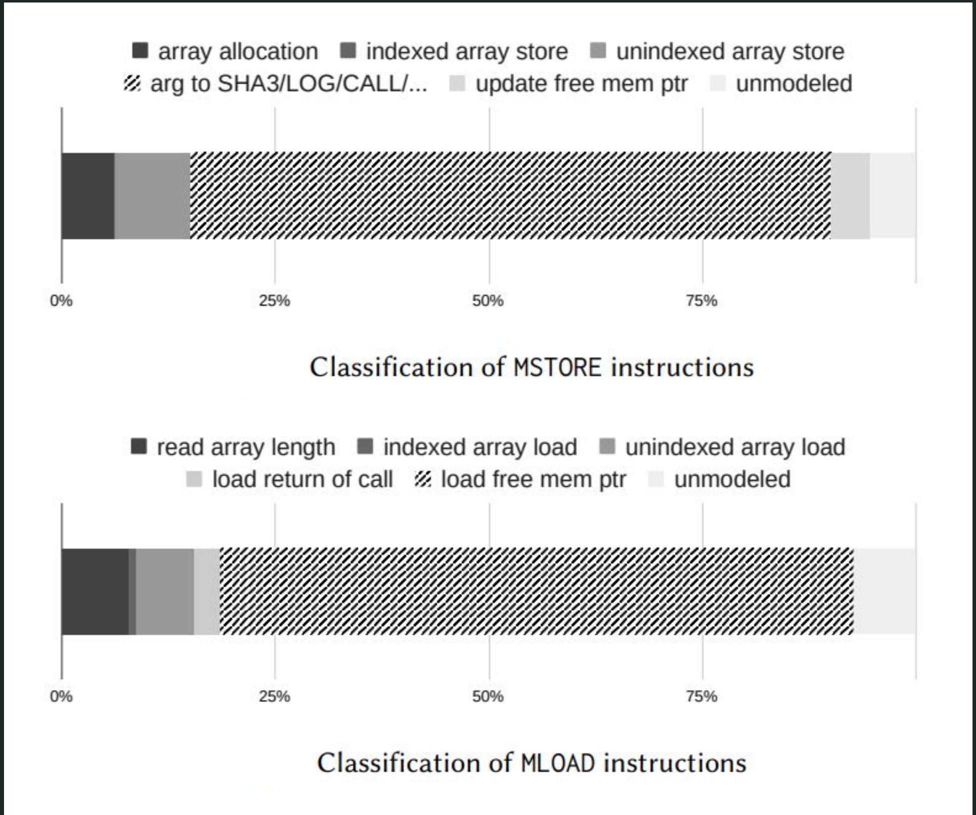
Matches MSTOREs that write for a memory consuming statement to the relative offset they write to

Client Analyses

The next 2 analyses require **memory modeling**, as they uses operations that are done through non-trivial uses of memory.

- Passing the arguments to an external call
- Getting its return value back

Client Analyses - Evaluation



94,59%

92,72%

Client Analysis - Taint Analysis

Tainted ERC20 Token transfer

- The transfer(address **recipient**, uint256 **amount**) function(part of the ERC20 interface) when called transfers **amount** tokens. The tainted ERC20 token transfer is **vulnerable!**

```
contract Victim {
  address owner ;
  function init () public {
      owner = msg . sender ; ... }
  function withdrawTokens ( address _tokenContract ) public returns ( bool ) {
      require ( msg . sender == owner );
      Token token = Token ( _tokenContract );
      uint256 amount = token . balanceOf ( address (this));
      return token . transfer ( owner , amount );}
}
```

allows anyone to take over the contract's ownership and bypass the guard

Client Analysis - Taint Analysis

Introduction of new Guard Conditions

- Are introduced new guard conditions that use an external contract as a source of authority, calling it to approve or reject an attempt to call a sensitive operation.

```
contract Guarded {
    address auth = ...;
    function sensitiveOperation () public {
        require ( msg . sender == auth . owner () );
        ...
    }
    function otherSensitiveOperation () public {
        require ( auth . isOwner ( msg . sender ));
        ...
    }
}
```

Client Analysis - Taint Analysis

Manual inspection for the *tainted ERC20 token* `transfer` vulnerability.

MD5	LOC	TP/FP	Comment
7eacf	1441	0 / 1	requires sender to destroy token
c09fb	146	0 / 1	can only be used to send to untainted investors
cee49	244	1 / 0	composite
486df	490	0 / 1	unrecognized guard
92a49	511	0 / 1	unrecognized guard
17c8f	64	1 / 0	by design, airdrop
b1092	201	1 / 0	by design, airdrop
a4f0e	52	1 / 0	by design, airdrop
8bfbd	479	0 / 1	unrecognized guard
af93f	264	1 / 0	composite
f02a4	204	1 / 0	by design, airdrop
b30d4	1069	1 / 0	composite
78fcb	64	1 / 0	by design, airdrop
82815	652	1 / 0	composite
6ecdb	864	0 / 1	complex logic, tokens sent will be compensated
394d2	394	0 / 1	unrecognized guard
e3129	429	0 / 1	unrecognized guard
4f9ac	56	0 / 1	requires caller to transfer tokens first
29976	224	1 / 0	composite
95b19	237	1 / 0	composite
503f3	698	0 / 3	unrecognized guard
0f8ab	268	1 / 0	composite
040e6	599	0 / 1	unrecognized guard
fb0ae	227	1 / 0	composite
33350	74	1 / 0	unguarded transfer
Total:	9951	14 / 13	

Client Analysis - Precise Gas Consumption

- EIP-1884, part of the Istanbul hard fork, repriced many resource intensive operations which led to the increasing cost of SLOADs (200 to 800 gas), causing **fallback functions** (2300 gas limit) to fail.

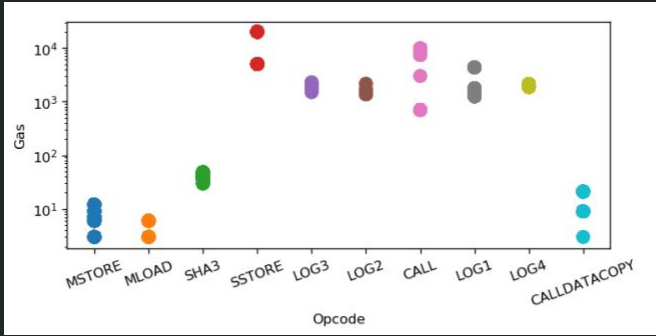
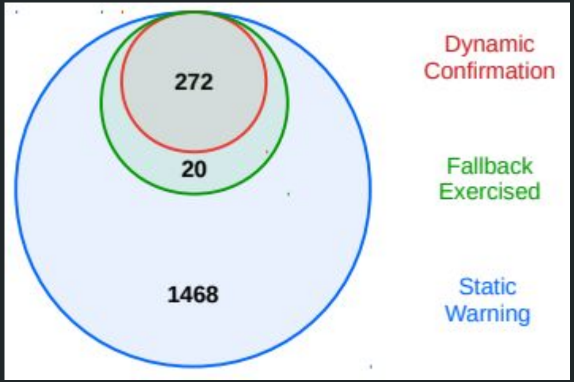


Image 1: *gas costs per opcode vary by orders of magnitude*

unnamed external function without any input or output parameters

- Smart contracts can’t have code whose execution cost can be easily manipulated by user input.**(instructions that use memory have a variable gas price!)**

Client Analysis - Precise Gas Consumption



- 93% precision of the gas analysis, meant to find fallback functions that would fail after EIP-1884 repricing

Client Analysis - Repeated Calls

Pattern: Two identical external calls, with one preceding the other.

same callee contract, target method and arguments

Why should it be avoided?

- Same external call could return different values
- Call cost gas

Evaluation:

Precise Static modeling analysis	Securify
85.96%	16.09%

Only constant memory modeling

Client Analysis - Repeated Calls (example)

```
interface Untrusted{
   function getBenificiary() external returns (address payable);
}

contract Victim{
   function isFriend (address addr) private returns (bool){(...)}

   function givEth(Untrusted untrustedAddress) public{
       if (isFriend (untrustedAddress.getBenificiary())){
           untrustedAddress.getBenificiary().transfer(1000 ether);
       }
   }
}
```

Might be exploited returning 2 different values