SYMBOLIC PROGRAM SLICING ON SMART CONTRACTS

Zi Xuan Chen, Fang Yu

National Chengchi University
Department of Computer Science, Management Information Systems
104703010@nccu.edu.tw, yuf@nccu.edu.tw

ABSTRACT

We propose a method to do program slicing on stack-based programming language. With slicing, we can analyse properties more efficiently. We take the EVM bytecode[1] as the example language, which is used to write smart contract[2] on Ethereum[1].

Keywords— BlockChain, Ethereum, Slicing, Verification

1. INTRODUCTION

There are many interest properties about Ethereum. Ethereum can be viewed as a transaction-based state machine. We can transit the transation state by sending transation or execute smart contract. On Ethereum, all transaction executed on Ethereum Virtual Machine (EVM), which is a simple stack-based architecture, limits stack item size to 1024. Executions will be failed if stackoverflow occured while executing the smart contract. Besides stack, ***gas*** is another interesting property on Ethereum. To limit the cost of the transaction execution, EVM takes the handling fee named ***gas*** from transaction sender.

To analyze the properties more precisely, we slice the smart contract by construcing dependencies between instructions. With the dependencies, we can slice a smaller program from some interested point. Then we can the the sliced program for other analysis purpose.

2. RELATED WORK

TBC.

3. METHOD

To compute the dependencies between instructions, we need construct the control flow graph first. Unlike register-based machine's instruction, which's operand are called as register explicitly, for the stack-based machine, the operand that instructions depended are stored on the stack implicitly. Thus, a control flow graph for stack simulation is needed.

The first step to construct control flow graph is spliting basic blocks. We split basic blocks by ***JUMPDEST*** and ***end instructions***. ***JUMPDEST*** is normally considered as the beginning of blocks becuase other blocks can target ***JUMPDEST*** to connect the edges. The ***end instructions*** include ***STOP***, ***SELFDESTRUCT***, ***RETURN***, ***REVERT***, ***INVALID***, ***SUICIDE***, ***JUMP***, ***JUMPI***.

The second step is building the edges between basic blocks. Some edges can be computed by simply succeeding the pc of the ***JUMPI*** and other instructions $\notin$ ***end instructions***, followed by ***JUMPDEST***. Because the property of the stack-based machine, all the jump destinations are pushed to stack implicitly. For this problem, we use value set analysis to find all the possible destination values.

Most of smart contracts are written as Solidity, whcih is an object-oriented (or even called contract-oriented), high-level language for implementing smart contracts. The contract in Solidity is like an object. Users can call the public functions in the contract, which we can treat as member functions in a object. From lower-level — EVM bytecode, the Solidity compiler will compile a dispatcher to dispatch all the public functions in the contract. The dispatcher can recognize the function hashes in transactions that users sent via Application Binary Interface (ABI). We compute the function boundaries and apply the value set analysis on each function to construct complete contract flow graph and instruction dependencies.

In the value set analysis (VSA), we traverse the basic blocks in control flow graph and continue finding new target address of next block by simulate the execution of instructions with a abstract stack. Abstract stack abstracts all the possible statck state. The nth-item in abstract stack represent a set of all possible value of nth-item in all possible stack state. So it is a over-approximation to compute the jump destinations. For each block, we record the states of the abstract stack before and after executing the instructions in the block. With the states, we can check the converge of the analysis. If we revisit a block, and it's post-execution state is same as last time, we consider it achieve the converge. We assume no new value would be found. Note that only the ***PUSH***, ***SWAP***, ***DUP***, ***AND*** instruction are implemented here, for other instructions, we only do the push, pop on the stack based on the operation times it defined. It's because other instructions im-

plementation will not affect the target address computation.

The instructions dependencies are also constructed while doing value set analysis. To build the dependencies between instructions, we need keeping track of the data flow of instruction operands. All the operands are pushed into and poped from the stack. Instead of marking all the operands with the instructions which pushed it, we push the instructions to the stack directly, and edges are added when the instructions are popped. Note that we don't use abstract stack here.Instead, we use a set of stack to keep all the states of possible stacks to maintain the accuracy of each operand list. If we use abstract stack here, more combination of operand list will be generated. It will lead more ambiguous result for address evaluation while building memory dependency.

After building the instruction dependency with stacks by value set analysis. we start to build the dependency between instruction by address of memory and storage.

4. EXPERIMENT

5. CONCLUSION

6. REFERENCES

[1] Gavin Wood, "Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07)," 2017, Accessed: 2018-01-03.

[2] Nick Szabo, "Advances in distributed security," 2003, Accessed: 2016-04-31.

Algorithm 1: buildCFGandStackDependency

```
   Input: Opcode
   Output: CFG, StackDG
 1 function buildCFGandStackDependency(opcode)
 2     dg = DG(opcode);
 3     cfg = CFG(opcode);
 4     cfg.buildBasicBlocks();
 5     cfg.buildSimpleEdges();
 6     cfg.buildFunctions(cfg.basicBlocks.first);
 7     for func ∈ cfg.functions do
 8         valueSetAnalysis(cfg, dg, func);
 9     end
10     return cfg, dg;
11 end function
   /* state constructor */
12 function State()
13     this.visit = dictionary(default=0);
14     this.stacksIn = dictionary(default=None);
15     this.stacksOut = dictionary(default=None);
16     this.lastDiscoveredTargets =
         dictionary(default=∅);
17     this.discoveredTargets = dictionary(default=∅);
18 end function
   /* value set analysis */
19 function valueSetAnalysis(cfg, dg, func)
20     stat = State();
21     toExplore = { func.entry };
22     do
23         outBlocks = { toExplore.pop() };
24         do
25             block = outBlocks.pop();
26             outBlocks = outBlocks ∪
27                 transFuncBlock(cfg, dg, func,
28                                block, stat);
29         while outBlocks;
30         for src, dsts ∈ stat.lastDiscoveredTargets do
31             cfg.addEdges(src, dsts);
32             toExplore = toExplore ∪ dsts;
33         end
34         stat.lastDiscoveredTargets =
             dictionary(default=∅);
35     while toExplore;
36     cfg.computeReachability(func.entry, func.id);
37 end function
```

Algorithm 2: ValueSetAnalysisUtility

```
   /* trasfer function blocks */
1  function transFuncBlock(cfg, dg, func, block, stat)
2      if (func.id = DISPATCHER_ID
3              and block.reacheable)
4        or stat.visit[block] > visitLimit then
5          return;
6      stat.visit[block] += 1;
       /* save previous stack to check
          convergence */
7      prevStack, _ = stat.stacksOut[block]
8      oprdStack, instStack = abstStack(), listStack();
9      inBlocks = [b ∈ block.inBlocks
10                 | stat.stacksOut[b] ≠ None]
11     for father ∈ inBlocks do
12         ostk, istk = stat.stacksOut[father];
13         oprdStack = oprdStack.merge(ostk);
14         instStack = oprdStack.merge(istk);
15     end
       /* explore the block  */
16     exploreBlock(dg, block,
17             oprdStack, instStack, stat);
       /* add branch according the result  */
18     if block.end ∈ {JUMP, JUMPI} then
19         oprdStack, _ = stat.stacksIn[end];
20         for dst ∈ oprdStack.top().vals() do
21             if isJumpDest(dst) then
22                 addBranch(src, dst, stat);
23         end
24     oprdStack, _ = stat.stacksOut[end];
25     if prevStack ≠ oprdStack then
           /* not converged  */
26         return block.outBlocksByFunc(func.id);
27     return ∅;
28 end function
   /* explore basic block */
29 function exploreBlock(dg, block, oprdStack,
     instStack, stat)
30     for inst ∈ block.instructions do
31         stat.stacksIn[inst] = (oprdStack, instStack);
32         stat.stacksOut[inst] =
33           transferFuncInst(dg, inst,
34                 oprdStack, instStack, stat);
35     end
36 end function
   /* add branch to value set analysis */
37 function addBranch(src, dst, stat)
38     if dst ∉ stat.discoveredTargets[src] then
39         if src ∉ stat.lastDiscoveredTargets then
40             stat.lastDiscoveredTargets[src] = ∅;
41         stat.lastDiscoveredTargets[src].add(dst);
42         stat.discoveredTargets[src].add(dst);
43 end function
```

Algorithm 3: ValueSetAnalysisUtility

```
   /* transfer instruction  */
1  function transferFuncInst(dg, inst, oprdStack, instStack,
     stat)
2      oprdStack = oprdStack.copy();
3      instStack = instStack.copy();
4      if inst ∈ PUSHn[n=1..32] then
5          oprdStack.push(inst.operand);
6          instStack.push(inst);
7      else if inst ∈ SWAPn[n=1..16] then
8          oprdStack.swap(n);
9          instStack.swap(n);
10     else if inst ∈ DUPn[n=1..16] then
11         oprdStack.dup(n);
12         instStack.dup(n);
13     else if inst = AND then
14         v1, v2 = oprdStack.pop(), oprdStack.pop();
15         oprdStack.push(absAnd(v1, v2));
16         v1s, v2s = instStack.pop(), instStack.pop();
17         for v1, v2 ∈ zip(v1s, v2s) do
18             dg.addEdges(inst, [v1, v2]);
19         end
20         instStack.push(inst);
21     else
22         repeat inst.popNumber times
23             oprdStack.pop();
24         end
25         for args ∈ [instStack.pop()
26                 | n ∈ range(inst.popNumber)]^T do
27             dg.addEdges(inst, args);
28         end
29         repeat inst.pushNumber times
30             oprdStack.push(None);
31             instStack.push(inst);
32         end
33     return oprdStack, instStack;
34 end function
```

Algorithm 4: AnalysisEnvironment

```
   /* Return All dependant program counters */
1  function depInsts(env, inst)
2      return env.addrDepInsts[inst]
3             ∪ env.offsetDepInsts[inst]
4             ∪ env.valDepInsts[inst]
5  end function
   /* check insts if have same addr parameters  */
6  function addrOverlap(env, instA, instB)
7      rangeA = product(env.conAddrs[instA],
8                       env.conOffsets[instA]);
9      rangeB = product(env.conAddrs[instB],
10                      env.conOffsets[instB]);
11     for (Aa, Ao), (Ba, Bo) ∈ product(rangeA, rangeB) do
12         if {Aa..Aa + Ao} ∩ {Ba..Ba + Bo} ≠ ∅ then
13             return True;
14     end
15     return False;
16 end function
   /* Environment constructor */
17 function Environment(stackDg)
18     rInsts = stackDg.rInsts ;
19     wInsts = stackDg.wInsts ;
20     addrs = [ i, eval(i.addrs) | i ∈ rInsts ∪ wInsts ] ;
21     offsets = [ i, eval(i.offsets) | i ∈ rInsts ∪ wInsts ] ;
22     vals = [ i, eval(i.vals) | i ∈ wInsts ] ;
       /* eval instructions' parameters (addr) */
23     this.conAddrs = { i: con | (i, (con, _)) ∈ addrs };
24     this.addrDepInsts = { i: dep | (i, (_ , dep)) ∈ addrs };
       /* eval instructions' parameters (offset) */
25     this.conOffsets = { i: con | (i, (con, _)) ∈ offsets };
26     this.offsetDepInsts = { i: dep | (i, (_ , dep)) ∈ offsets };
       /* eval instructions' parameters (val) */
27     this.conVals = { i: con | (i, (con, _)) ∈ vals };
28     this.valDepInsts = { i: dep | (i, (_ , dep)) ∈ vals };
       /* write insts that can't be re-evaled */
29     this.evaled = {i ∈ addrDepInsts | addrDepInsts[i] = ∅}
30       ∩ {i ∈ offsetDepInsts | offsetDepInsts[i] = ∅}
31       ∩ {i ∈ valDepInsts | valDepInsts[i] = ∅}
32 end function
```

Algorithm 5: buildAddressDependency

```
   Input: CFG, StackDG
   Output: AddressDG
1  import AnalysisEnvironment as env
2  function buildAddressDependency(cfg, stackDg, opcode)
       /* declare and alias variables */
3      addrDg = DG(opcode);
4      visit, alter = ∅, ∅ ;
5      sreads = stackDg.SLOADs ;
6      swrites = stackDg.SSTOREs ;
7      mreads = stackDg.MSTOREs ;
8      mwrites = stackDg.{MLOAD ∪ SHA3
9                ∪ CREATE ∪ CALL ∪ RETURN}s
       /* new a environment */
10     env = Environment(stackDg);
       /* build addr dependency of  */
       /* storage and memroy */
11     while True do
12         evaled = env.evaled.copy();
13         buildDependency(addrDg, swrites, sreads, visit);
14         buildDependency(addrDg, mwrites, mreads, visit);
15         if exist write inst can be re-evaled then
16             for inst ∈ re-evaled do
17                 update Environment variables in env
18                     with eval(instruction parameters)
19             end
20         if env.evaled \ evaled = ∅ then
21             break;
22     end
23     return addrDg;
24 end function
   /* helper function */
25 function buildDependency(addrDg, writes, reads, visit)
26     concrete = {inst ∈ (writes \ visit)
27                 | depInsts(env, inst) = ∅};
28     while concrete ≠ ∅ do
29         for inst ∈ concrete do
30             block = CFG.blockOf(inst);
31             dfsCFG(addrDg, inst, block, writes, reads, ∅);
32         end
33         visit.update(concrete);
34         for inst ∈ (writes \ visit) do
35             if (depInsts(env, inst) \ env.evaled) = ∅ then
36                 update Environment variables in env
37                     with eval(instruction parameters)
38         end
39         concrete = {ins ∈ (writes \ visit)
40                     | depInsts(env, ins) = ∅};
41     end
42 end function
```

Algorithm 6: dfsCFG

```
1  import AnalysisEnvironment as env
   /* do CFG dfs for building dependency */
2  function dfsCFG(addrDg, wInst, block, writes, reads, visit)
3      visit.add(block);
4      rwInsts = block.insts ∩ (writes ∪ reads);
5      if wInst ∈ block then
6          rwInsts = rwInsts \
7              { inst ∈ block.insts | inst.pc < wInst.pc};
8      for inst ∈ rwInsts do
           /* if "exist the probability" to re-write the
              same address then return, "probability"
              means the "or" part */
9          if inst.name = wInst.name
10             and (addrOverlap(env, wInst, inst)
11                 or env.addrDepInsts[inst] ≠ ∅) then
12             visit.remove(block);
13             return;
14         if inst ∈ reads then
15             deps = env.addrDepInsts[inst] ∪
16                     env.offsetDepInsts[inst];
17             if deps ≠ ∅ and
18               deps \ env.evaled = ∅ then
19                 update Environment variables in env
20                     with eval(instruction parameters)
21             if addrOverlap(env, wInst, inst) then
22                 env.evaled.add(inst);
23                 addrDg.addEdge(wInst, inst);
24                 env.conVals[inst].update(
25                                 env.conVals[wInst]);
26     end
27     for nextBlock ∈ block.outBlock do
28         dfsCFG(wInst, nextBlock, writes, reads, visit);
29     end
30     visit.remove(block);
31 end function
```

Algorithm 7: eval

Input: $instruction$, $visit$
Output: $concrete\ values$, $dependant\ PCs$

```
1  function eval(inst, visit)
2      if inst ∈ visit then
3          return ∅, {inst};
4      visit.add(inst);
5      concrete, dependant = ∅, ∅;
6      if inst.name.startswith('PUSH') then
7          return {int(op.operand)}, ∅;
8      cons, deps = {map(eval(_ , visit), argList)
9                          | argList ∈ inst.argLists}^T;
10     for argList ∈ cons do
11         val = None;
12         if None ∈ argList then
13             continue;
14         else if inst.name = 'ADD' then
15             val = let x, y = argList in x + y;
16         else if inst.name = 'SUB' then
17             val = let x, y = argList in x - y;
18         else if inst.name = 'MUL' then
19             val = let x, y = argList in x * y;
20         else if inst.name = 'DIV' then
21             val = let x, y = argList in x / y;
22         else if inst.name = 'EXP' then
23             val = let x, y = argList in x^y;
24         else if inst.name = 'ISZERO' then
25             val = let[x] = argList in { 0, if x = 0
                                         { 1, otherwise
26         else if inst.name = 'NOT' then
27             val = let [x] = argList in (1 << 256) - 1 - x;
28         else if inst.name = 'AND' then
29             val = let x, y = argList in x & y;
30         else if inst.name = 'OR' then
31             val = let x, y = argList in x | y;
32         else if inst.name = 'EQ' then
33             val = let x, y = argList in x == y;
34         else if inst.name ∈ {'MLOAD', 'SLOAD', 'SHA3'}
             then
35             concrete.update(env.conVals[inst]);
36         else
               /* SHA3 not impl yet */
37             throw Exception("not handle the inst yet");
38         if val ≠ None then
39             concrete.add(val);
40         dependant.update(concat(deps));
41     end
42     visit.remove(inst);
43     return concrete, dependant;
44 end function
```