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. 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.

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 prev 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
16     exploreBlock(dg, block, oprdStack, instStack, stat);
17     if block.end ∈ {JUMP, JUMPI} then
18         oprdStack, _ = stat.stacksIn[end];
19         for dst ∈ oprdStack.top().vals() do
20             if isJumpDest(dst) then
21                 addBranch(src, dst, stat);
22         end
23     oprdStack, _ = stat.stacksOut[end];
24     if prevStack ≠ oprdStack then
           /* not converged */
25         return block.outBlocksByFunc(func.id);
26     return ∅;
27 end function
   /* explore basic block */
28 function exploreBlock(dg, block, oprdStack, instStack, stat)
29     for inst ∈ block.instructions do
30         stat.stacksIn[inst] = (oprdStack, instStack);
31         stat.stacksOut[inst] =
32           transferFuncInst(dg, inst,
33                 oprdStack, instStack, stat);
34     end
35 end function
   /* add branch to value set analysis */
36 function addBranch(src, dst, stat)
37     if dst ∉ stat.discoveredTargets[src] then
38         if src ∉ stat.lastDiscoveredTargets then
39             stat.lastDiscoveredTargets[src] = ∅;
40         stat.lastDiscoveredTargets[src].add(dst);
41         stat.discoveredTargets[src].add(dst);
42 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
import AnalysisEnvironment as env
function buildAddressDependency(cfg, stackDg, opcode)
    /* declare and alias variables */
    addrDg = DG(opcode);
    visit, alter = ∅, ∅ ;
    sreads = stackDg.SLOADs ;
    swrites = stackDg.SSTOREs ;
    mreads = stackDg.MSTOREs ;
    mwrites = stackDg.{MLOAD ∪ SHA3
                  ∪ CREATE ∪ CALL ∪ RETURN}s
    /* new a environment */
    env = Environment(stackDg);
    /* build addr dependency of  */
    /* storage and memroy */
    while True do
        evaled = env.evaled.copy();
        buildDependency(addrDg, swrites, sreads, visit);
        buildDependency(addrDg, mwrites, mreads, visit);
        if exist write inst can be re-evaled then
            for inst ∈ re-evaled do
                update Environment variables in env
                    with eval(instruction parameters)
            end
        if env.evaled \ evaled = ∅ then
            break;
    end
    return addrDg;
end function
/* helper function */
function buildDependency(addrDg, writes, reads, visit)
    concrete = {inst ∈ (writes \ visit)
                  | depInsts(env, inst) = ∅};
    while concrete ≠ ∅ do
        for inst ∈ concrete do
            block = CFG.blockOf(inst);
            dfsCFG(addrDg, inst, block, writes, reads, ∅);
        end
        visit.update(concrete);
        for inst ∈ (writes \ visit) do
            if (depInsts(env, inst) \ env.evaled) = ∅ then
                update Environment variables in env
                    with eval(instruction parameters)
        end
        concrete = {ins ∈ (writes \ visit)
                      | depInsts(env, ins) = ∅};
    end
end function
```

Algorithm 6: dfsCFG

```
import AnalysisEnvironment as env
/* do CFG dfs for building dependency */
function dfsCFG(addrDg, wInst, block, writes, reads, visit)
    visit.add(block);
    rwInsts = block.insts ∩ (writes ∪ reads);
    if wInst ∈ block then
        rwInsts = rwInsts \
            { inst ∈ block.insts | inst.pc < wInst.pc};
    for inst ∈ rwInsts do
        /* if "exist the probability" to re-write the
           same address then return, "probability"
           means the "or" part */
        if inst.name = wInst.name
          and (addrOverlap(env, wInst, inst)
              or env.addrDepInsts[inst] ≠ ∅) then
            visit.remove(block);
            return;
        if inst ∈ reads then
            deps = env.addrDepInsts[inst] ∪
                      env.offsetDepInsts[inst];
            if deps ≠ ∅ and
              deps \ env.evaled = ∅ then
                update Environment variables in env
                    with eval(instruction parameters)
            if addrOverlap(env, wInst, inst) then
                env.evaled.add(inst);
                addrDg.addEdge(wInst, inst);
                env.conVals[inst].update(
                                env.conVals[wInst]);
    end
    for nextBlock ∈ block.outBlock do
        dfsCFG(wInst, nextBlock, writes, reads, visit);
    end
    visit.remove(block);
end function
```

Algorithm 7: eval

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

function *eval(inst, visit)*
  if *inst* $\in$ *visit* **then**
    return $\emptyset$, {inst};
  visit.add(inst);
  concrete, dependant = $\emptyset$, $\emptyset$;
  if *inst.name.startswith('PUSH')* **then**
    return {int(op.operand)}, $\emptyset$;
  cons, deps = {map(eval(_ , visit), argList)
                | argList $\in$ inst.argLists}^{T};
  for *argList* $\in$ *cons* **do**
    val = None;
    if *None* $\in$ *argList* **then**
      continue;
    else if *inst.name* = *'ADD'* **then**
      val = **let** x, y = argList **in** x + y;
    else if *inst.name* = *'SUB'* **then**
      val = **let** x, y = argList **in** x - y;
    else if *inst.name* = *'MUL'* **then**
      val = **let** x, y = argList **in** x * y;
    else if *inst.name* = *'DIV'* **then**
      val = **let** x, y = argList **in** x / y;
    else if *inst.name* = *'EXP'* **then**
      val = **let** x, y = argList **in** x^y;
    else if *inst.name* = *'ISZERO'* **then**
      $$val = \textbf{let}[x] = argList\ \textbf{in} \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{otherwise} \end{cases}$$
    else if *inst.name* = *'NOT'* **then**
      val = **let** [x] = argList **in** (1 << 256) - 1 - x;
    else if *inst.name* = *'AND'* **then**
      val = **let** x, y = argList **in** x & y;
    else if *inst.name* = *'OR'* **then**
      val = **let** x, y = argList **in** $x \mid y$;
    else if *inst.name* = *'EQ'* **then**
      val = **let** x, y = argList **in** $x = y$;
    else if *inst.name* $\in$ {*'MLOAD', 'SLOAD', 'SHA3'*} **then**
      concrete.update(env.conVals[inst]);
    else
      `/* SHA3 not impl yet */`
      throw Exception("not handle the inst yet");
    if *val* $\neq$ *None* **then**
      concrete.add(val);
    dependant.update(concat(deps));
  end
  visit.remove(inst);
  return concrete, dependant;
end function