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] 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: AnalysisEnvironment

```
Input: StackDG
/* Environment variables */
evaled = ∅;
conAddrs = dictionary();
addrDepPCs = dictionary();
conOffsets = dictionary();
offsetDepPCs = dictionary();
conVals = dictionary();
valDepPCs = dictionary();
rwDependency = dictionary();
/* Return All dependant program counters */
function depPCs(inst)
    return addrDepPCs[inst.pc]
            ∪ offsetDepPCs[inst.pc]
            ∪ valDepPCs[inst.pc]
end function
/* check insts if have same addr parameters  */
function addrOverlap(insA, insB)
    rangeA = product(conAddrs[insA.pc],
                     conOffsets[insA.pc]);
    rangeB = product(conAddrs[insB.pc],
                     conOffsets[insB.pc]);
    for (Aa, Ao), (Ba, Bo) ∈ product(rangeA, rangeB) do
        if {Aa..Aa + Ao} ∩ {Ba..Ba + Bo} ≠ ∅ then
            return True;
    end
    return False;
end function
/* Initialize Environment */
function initialize()
    global Environment variables;
    readIns = StackDG.readIns ;
    writeIns = StackDG.writeIns ;
    /* eval instructions' parameters (addr) */
    addrs = [ i.pc, eval(i.addrs) | i ∈ readIns ∪ writeIns ] ;
    conAddrs = { pc: con | (pc, (con, _)) ∈ addrs };
    addrDepPCs = { pc: dep | (pc, (_ , dep)) ∈ addrs };
    /* eval instructions' parameters (offset) */
    offsets = [ i.pc, eval(i.offsets) | i ∈ readIns ∪ writeIns ] ;
    conOffsets = { pc: con | (pc, (con, _)) ∈ offsets };
    offsetDepPCs = { pc: dep | (pc, (_ , dep)) ∈ offsets };
    /* eval instructions' parameters (val) */
    vals = [ i.pc, eval(i.vals) | i ∈ writeIns ] ;
    conVals = { pc: con | (pc, (con, _)) ∈ vals };
    valDepPCs = { pc: dep | (pc, (_ , dep)) ∈ vals };
    /* write insts that can't be re-evaled */
    evaled = {pc ∈ addrDepPCs | addrDepPCs[pc] = ∅}
      ∩ {pc ∈ offsetDepPCs | offsetDepPCs[pc] = ∅}
      ∩ {pc ∈ valDepPCs | valDepPCs[pc] = ∅}
end function
```

Algorithm 2: buildAddressDependency

```
Input: CFG, StackDG
Output: AddressDG
import AnalysisEnvironment as env
function buildAddressDependency()
    /* initialize the environment */
    env.initialize();
    /* declare and alias variables */
    visit, alter = ∅, ∅ ;
    sreads = StackDG.SLOADs ;
    swrites = StackDG.SSTOREs ;
    mreads = StackDG.MSTOREs ;
    mwrites = StackDG.{MLOAD ∪ SHA3
                ∪ CREATE ∪ CALL ∪ RETURN}s
    /* build addr dependency of  */
    /* storage and memroy */
    while True do
        evaled = env.evaled.copy();
        buildDepend(swrites, sreads, visit);
        buildDepend(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 env.rwDependency;
end function
/* helper function */
function buildDepend(writes, reads, visit)
    concrete = {ins ∈ (writes \ visit)
                    | env.depPCs(ins) = ∅};
    while concrete ≠ ∅ do
        for inst ∈ concrete do
            block = CFG.blockOf(inst);
            dfsCFG(inst, block, writes, reads, ∅);
        end
        visit.update(concrete);
        for inst ∈ (writes \ visit) do
            if (env.depPCs(inst) \ env.evaled) = ∅ then
                update Environment variables in env
                    with eval(instruction parameters)
        end
        concrete = {ins ∈ (writes \ visit)
                        | env.depPCs(ins) = ∅};
    end
end function
```

Algorithm 3: dfsCFG

```
Input: CFG, StackDG
Output: AddressDG
import AnalysisEnvironment as env
/* do CFG dfs for building dependency */
function dfsCFG(wInst, block, writes, reads, visit)
    visit.add(block);
    rwInsts = (block.insts ∩ writes ∩ reads);
    if wInst ∈ block then
        rwInsts = rwInsts \
            { ins ∈ block.insts | ins.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
         (env.addrOverlap(wInst, inst)
              or env.addrDepPCs[inst.pc] ≠ ∅) then
            visit.remove(block);
            return;
        if inst ∈ reads then
            depPCs = env.addrDepPCs[inst.pc] ∪
                     env.offsetDepPCs[inst.pc];
            if depPCs ≠ ∅ and
               depPCs \ env.evaled = ∅ then
                update Environment variables in env
                    with eval(instruction parameters)
            if addrOverlap(wInst, inst) then
                env.evaled.add(pc);
                env.rwDependency[inst.pc].add(wInst);
                env.conVals[inst.pc].update(
                                env.conVals[wInst.pc]);
    end
    for nextBlock ∈ block.outgoingBlock do
        dfsCFG(wInst, nextBlock, writes, reads, visit);
    end
    visit.remove(block);
end function
```

Algorithm 4: eval

```
Input: StackDG, instruction, visit
Output: concrete values, dependant PCs
function eval(inst, visit)
    if inst ∈ visit then
        return {}, {inst};
    visit.add(inst);
    concrete, dependant = ∅, ∅;
    if inst.name.startswith('PUSH') then
        return {int(op.operand)}, {};
    cons, deps = {map(eval(_ , visit), argList)
                        | argList ∈ inst.argLists}^T;
    for argList ∈ cons do
        val = None;
        if None ∈ 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 = let[x] = argList in { 0, if x = 0
                                      { 1, otherwise
        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 | y;
        else if inst.name = 'EQ' then
            val = let x, y = argList in x = y;
        else if inst.name ∈ {'MLOAD', 'SLOAD', 'SHA3'}
         then
            concrete.update(env.conVals[inst.pc]);
        else
            /* SHA3 not impl yet */
            trhow Excption("not handle the inst yet");
        if val ≠ None then
            concrete.add(val);
        dependant.update(concat(deps));
    end
    visit.remove(inst);
    return concrete, dependant;
end function
```