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. The word size of the machine (and thus size of stackitems) is 256-bit. 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 dependency graph (DG). 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 (CFG) 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 CFG for stack simulation is needed.

Algorithm 1: buildCFGandStackDependency

```
Input: Opcode
Output: CFG, StackDG
function buildCFGandStackDependency(opcode)
    dg, cfg = DG(opcode), CFG(opcode);
    cfg.buildBasicBlocks();
    cfg.buildSimpleEdges();
    cfg.buildFunctions(cfg.basicBlocks.first);
    for func ∈ cfg.functions do
        valueSetAnalysis(cfg, dg, func);
    end
    return cfg, dg;
end function
/* state constructor */
function State()
    this.visit = dict(dflt=0);
    this.stacksIn = dict(dflt=None);
    this.stacksOut = dict(dflt=None);
    this.discoveredTargets = dict(dflt=∅);
    this.lastDiscoveredTargets = dict(dflt=∅);
end function
/* value set analysis */
function valueSetAnalysis(cfg, dg, func)
    stat = State();
    toExplore = { func.entry };
    do
        outBlocks = { toExplore.pop() };
        do
            outBlocks = outBlocks ∪
                transFuncBlock(cfg, dg, func,
                    outBlocks.pop(), stat);
        while outBlocks;
        for src, dsts ∈ stat.lastDiscoveredTargets do
            cfg.addEdges(src, dsts);
            toExplore = toExplore ∪ dsts;
        end
        stat.visit = dict(dflt=0);
        stat.lastDiscoveredTargets = dict(dflt=∅);
    while toExplore;
end function
```

The first step to construct CFG 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***, ***SELF-DESTRUCT***, ***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 (VSA) to find all the possible destination values.

Most of smart contracts are written in Solidity, whcih is an object-oriented (or 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 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 VSA on each function to construct complete contract flow graph and instruction dependencies.

In the VSA, we traverse the basic blocks in CFG 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*** 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 implementation will not affect the target address computation.

The instructions dependencies are also constructed while doing VSA. To build the dependencies, we need keeping track of the data flow of operands. All the operands are pushed into and poped from the stack. Instead of marking 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.

Algorithm 2: ValueSetAnalysisUtility

```
/* trasfer function blocks */
function transFuncBlock(cfg, dg, func, block, stat)
    if (func.id = DISPATCHER_ID
            and block.reacheable)
      or stat.visit[block] > visitLimit then
        return;
    stat.visit[block] += 1;
    /* save pre-stack to check convergence */
    prevStack, _ = stat.stacksOut[block]
    oprdStack, instStack = abstStack(), listStack();
    inBlocks = [b ∈ block.inBlocks
                  | stat.stacksOut[b] ≠ None]
    for father ∈ inBlocks do
        ostk, istk = stat.stacksOut[father];
        oprdStack = oprdStack.merge(ostk);
        instStack = oprdStack.merge(istk);
    end
    /* explore the block */
    exploreBlock(dg, block,
          oprdStack, instStack, stat);
    /* add branch according the result */
    if block.end ∈ {JUMP, JUMPI} then
        oprdStack, _ = stat.stacksIn[end];
        for dst ∈ oprdStack.top().vals() do
            if isJumpDest(dst) then
                addBranch(src, dst, stat);
        end
    oprdStack, _ = stat.stacksOut[end];
    if prevStack ≠ oprdStack then
        /* not converged */
        return block.outBlocksByFunc(func.id);
    return ∅;
end function
/* explore basic block */
function exploreBlock(dg, block, oprdStack,
 instStack, stat)
    for inst ∈ block.instructions do
        stat.stacksIn[inst] = (oprdStack, instStack);
        stat.stacksOut[inst] = transferFuncInst(
                dg, inst, oprdStack, instStack, stat);
    end
end function
/* add branch to value set analysis */
function addBranch(src, dst, stat)
    if dst ∉ stat.discoveredTargets[src] then
        if src ∉ stat.lastDiscoveredTargets then
            stat.lastDiscoveredTargets[src] = ∅;
        stat.lastDiscoveredTargets[src].add(dst);
        stat.discoveredTargets[src].add(dst);
end function
```

Algorithm 3: ValueSetAnalysisUtility

```
/* transfer instruction  */
function transferFuncInst(dg, inst,
                    oprdStack, instStack, stat)
    oprdStack = oprdStack.copy();
    instStack = instStack.copy();
    if inst ∈ PUSHn[n=1..32] then
        oprdStack.push(inst.operand);
        instStack.push(inst);
    else if inst ∈ SWAPn[n=1..16] then
        oprdStack.swap(n);
        instStack.swap(n);
    else if inst ∈ DUPn[n=1..16] then
        oprdStack.dup(n);
        instStack.dup(n);
    else if inst = AND then
        v1, v2 = oprdStack.pop(), oprdStack.pop();
        oprdStack.push(absAnd(v1, v2));
        v1s, v2s = instStack.pop(), instStack.pop();
        for v1, v2 ∈ zip(v1s, v2s) do
            dg.addEdges(inst, [v1, v2]);
        end
        instStack.push(inst);
    else
        repeat inst.popTimes times
            oprdStack.pop();
        end
        for args ∈ [instStack.pop()
                | n ∈ range(inst.popTimes)]^T do
            dg.addEdges(inst, args);
        end
        repeat inst.pushTimes times
            oprdStack.push(None);
            instStack.push(inst);
        end
    return oprdStack, instStack;
end function
```

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.

Memory and storage are other two main data read/write mechanisms on Ethereum except stack, where the memory is volatile, the storage is non-volatile. By the way, according the figure below, we can find that the EVM does not follow the standard von Neu-mann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.[1]

There is an indexer recording current memory used, it indicates the maximum index of current memory used. The total fee for memory-usage payable is proportional to smallest

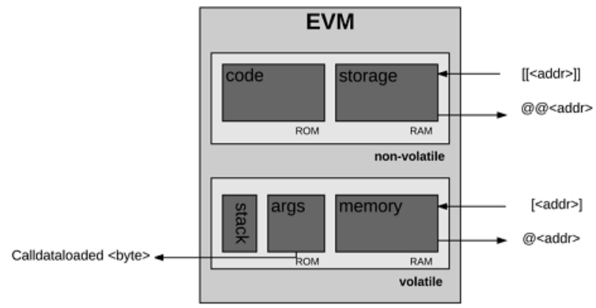

Fig. 1. EVM architecture

multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range[1].

Storage fees have a slightly nuanced behaviour—to incentivise minimisation of the use of storage (which corresponds directly to a larger state database on all nodes),the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given;in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more than normal usage[1].

The main idea to construct address dependency is traversing the CFG from write instructions with a adress, and build the dependencies with the instructions which read the address, until meet the next instructions whcih rewrite the address. With the DG we constructed, we can evaluate some address values and the stored values from known constants by the dependencies. There are some instructions about the read/write operation on memory and storage. For the storage, the write instructions is ***SSTORE***, the load instruction is ***SLOAD***. For the memory, the write instruction is ***MSTORE***, the read instruction include ***MLOAD***, ***SHA3***, ***CREATE***, ***CALL***, ***RETURN***.

For write instructions, there are three part we concerned: the address it stored, the range of memory (offset) it covered and the value it wrote. For each part, if we could eval it as constants from other constants (normally come from the terminal of DG — ***PUSH*** instruction), we save the values as a concrete value set for the part. if not, all the instruction that it needed to do evaluation, would be saved as a dependant instruction set, once all the instruction in the set are evaluated, the part could be evaluated again to get the concrete values. Same as the write instructions, the read instructions also have these part. But for the values it load, are depended on the write instructions' value which have the same address as it, that's just the dependency we want to construct.

Algorithm 4: AnalysisEnvironment

```
/* Return All dependant program counters */
function depInsts(env, inst)
    return env.addrDepInsts[inst]
           ∪ env.offsetDepInsts[inst]
           ∪ env.valDepInsts[inst]
end function
/* check insts if have same addr parameters */
function addrOverlap(env, instA, instB)
    rangeA = product(env.conAddrs[instA],
                     env.conOffsets[instA]);
    rangeB = product(env.conAddrs[instB],
                     env.conOffsets[instB]);
    for (Aa, Ao), (Ba, Bo) ∈ product(rangeA, rangeB) do
        if {Aa..Aa + Ao} ∩ {Ba..Ba + Bo} ≠ ∅ then
            return True;
    end
    return False;
end function
/* Environment constructor */
function Environment(stackDg)
    rInsts = stackDg.rInsts ;
    wInsts = stackDg.wInsts ;
    addrs = [ i, eval(i.addrs) | i ∈ rInsts ∪ wInsts ] ;
    offsets = [ i, eval(i.offsets) | i ∈ rInsts ∪ wInsts ] ;
    vals = [ i, eval(i.vals) | i ∈ wInsts ] ;
    /* eval instructions' parameters (addr) */
    this.conAddrs = { i: con | (i, (con, _)) ∈ addrs };
    this.addrDepInsts = { i: dep | (i, (_ , dep)) ∈ addrs };
    /* eval instructions' parameters (offset) */
    this.conOffsets = { i: con | (i, (con, _)) ∈ offsets };
    this.offsetDepInsts = { i: dep | (i, (_ , dep)) ∈ offsets };
    /* eval instructions' parameters (val) */
    this.conVals = { i: con | (i, (con, _)) ∈ vals };
    this.valDepInsts = { i: dep | (i, (_ , dep)) ∈ vals };
    /* write insts that can't be re-evaled */
    this.evaled = {i ∈ addrDepInsts | addrDepInsts[i] = ∅}
      ∩ {i ∈ offsetDepInsts | offsetDepInsts[i] = ∅}
      ∩ {i ∈ valDepInsts | valDepInsts[i] = ∅}
end function
```

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 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 ∈ 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]);
        else
            /* SHA3 not impl yet */
            throw Exception("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
```