PASS Project – Team 28

[Section One] State Path and Block Path

[Section 1-1] state_path used for structuring statement execution sequence
    state_path(id_after: SSA, id_before: SSA)
    id_after executes after id_before
[Section 1-2] block_path used for structuring block execution sequence(goto, branch, jump)
    block_path(id_after: SSA, id_before: SSA)
    id_after executes after id_before

[Section Two] Dependency and Jump Relationship

[Section 2-1] Direct Depend on Message Sender
    DirectdepMs(i: SSA)
[Section 2-2] Direct Depend on Argument
    DirectdepSSA(i: SSA, i2: SSA)
[Section 2-3] Direct Depend on SSA(in this case, maybe also depend on other values)
    DirectdepSSA(i: SSA, i2: SSA)
[Section 2-4] Only Depend on SSA(in this case, only depend on this value)
    OnlydepSSA(i: SSA, i2: SSA)
[Section 2-5] Implicit Dependency
ImplicitDependarg(id1: SSA, id2: SSA) // Implicitly Depends on argument
ImplicitDependtaint(id1: SSA, id2: SSA) // Implicitly Depends on tainted value
ImplicitDepend(id1: SSA, id2: SSA) // Implicitly Depends on some SSA
[Section 2-6] Tainted pass relationship by Load and store
.decl taintedpass(id1: SSA, id2: SSA, b1: Block, b2: Block)
    id1 from block b1
        is tainted because of
    id2 from block b2
[Section 2-7] Function call relationship
jump_transfer(a: SSA, idx: ArgIndex, bfrom: Block, bto: Block, t: Transfer)
    argument (a) with index (idx) from block (bfrom)
        jump by calling function of block (bto)
    and the transfer id is t
jump_transfer_srt_end(a: SSA, index: ArgIndex, bfrom: Block, bto: Block, t: Transfer)
    This is used to stucture relationship of stack function call, and only get relationship of start block and end block, eg:
        function1 a1--> function2 a2--> function3 a3--> function4 a4--> function5(with argument a5)
        The relationship is
        jump_transfer_srt_end(a5, 0, b1, b5, t)

[Section Three] Guard

    .decl guard(block: Block, id: SSA)
    block: Block id, id: the SSA that is guarded
[Section 3-1] basic guard
Case1 Both the left and right is msg.sender / depends on msg.sender
Case2 the left is msg.sender/ depends on msg.sender,, the right is not tainted & not implicitly depended on sth
Case3 the right is msg.sender/ depends on msg.sender, the left is not tainted & not implicitly depended on sth
case4 the left is msg.sender/ depends on msg.sender, the right is trusted due to functional call
case5 the right is msg.sender/ depends on msg.sender, the left is trusted due to functional call

case6 the left directly depends on the argument, the right is msg.sender/ depends on msg.sender
case7 the right directly depends on the argument, the left is msg.sender/ depends on msg.sender

[Section 3-2] Guard associated with functional call

Consider bop(i, left,right,_), we have:
  [Section 3-2-A] Call Guard: Either left or right is a return value
  [Section 3-2-B] Conditional Guard: bop(i, left,right,_) is a return value

[Section 3-2-A] Call Guard

Case 1: the first one is msg.sender/depend on msg.sender and the second one is a functional call (the return must be untainted)
Case 2: the first one is untainted and the second one is a functional call (msg.sender/depend on msg.sender)

[Section 3-2-B] Conditional Guard

Type A: the first part is msg.sender / depend on msg.sender,and the second depends on the argument
Type B: the first part is untainted, and the second depends on the argument
Type C: the first part is untainted, and the second part depends on msg.sender / depend on msg.sender
Type D: both the first and the second part depends on the argument

[Section 3-3] Variable Guard

Sometimes we have situation that whether one is a guard or not depends on a variable which depends on another block, we need variable guard to solve this.
EG:

```
        function set_a() public {
          a = msg.sender;
          require(msg.sender == b); // guard
        }

        function set_b() public {
          b = msg.sender;
          require(msg.sender == a); // guard
        }
```

[Section 3-4] Guard Passing

Case1: guard passing within block with direct relationship
Case2: guard passing between linked blocks
Case3: guard passing between unlinked blocks

[Section Four] Tainted

```
.decl taintedBefore(statement_id:SSA, var_id:SSA)
Before executing statement_id, var_id is tainted
.decl taintedAfter(statement_id:SSA, var_id:SSA)
After executing statement_id, var_id is tainted
.decl blockBefore(id_Block: Block, var_id:SSA)
Before executing statements in id_Block, var_id is tainted
.decl blockAfter(id_Block: Block, var_id:SSA)
After executing all statements in id_Block, var_id is tainted
```

[Section Five] Wrapping up all relationship

```
tainted_sinks(id) :- selfdestruct(id, A), taintedBefore(id, A),
blockStmt(b1, id), !final_guard(b1, A).
tainted_sinks(id) :- selfdestruct(id, A), ImplicitDepend(A, _),
blockStmt(b1, id), !final_guard(b1, A).
```

When we have selfdestruct(id, A), if A is tainted or have implicit dependency, and A is not guarded, we then consider id as tainted_sinks