18 NOVEMBER 2025

How to solve stack-too-deep errors: a Yul to EVM pipeline

Alejandro Hernández Cerezo

Complutense University of Madrid & Grey Project & The Costa Group

Joint Work With: Elvira Albert, Pablo Gordillo, Albert Rubio & the Solidity Team

AGENDA

Producing bytecode that handles EVM stack operations efficiently remains a longstanding challenge

1. The EVM vs other stack-based machines and the *stack-too-deep* problem
2. How the Static Single Assignment form (SSA) helps designing stack layouts
3. How can we generate bytecode that uses the stack efficiently
4. How to repair stack-too-deep accesses in arbitrary situations using SSA
5. Experimental Evaluation & Conclusions

What makes the EVM unique compared to other stack-based machines?

1. Most stack-based machines use the operational stack for computations and have registers for passing the variables along the Control-Flow Graph (CFG). Very limited when using the stack.

2. EVM is a pure stack machine with no registers, so either the stack or memory regions must be used for that purpose. It is very expressive when using the stack.

Stack-too-deep problems arise organically in the EVM when only using the stack to pass variables

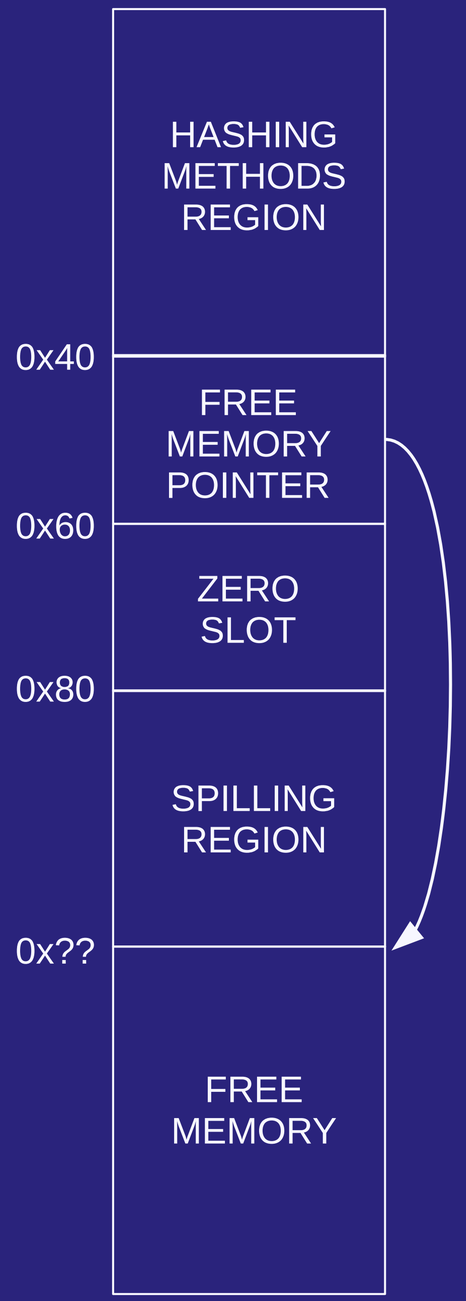
Context: a stack-too-deep access happen when trying to retrieve a variable from the stack that is beyond depth 16
(no DUPx or SWAPx available)

Situation: there are more than 16 variables that must be used (all of them) but also preserved in the stack for later computations.

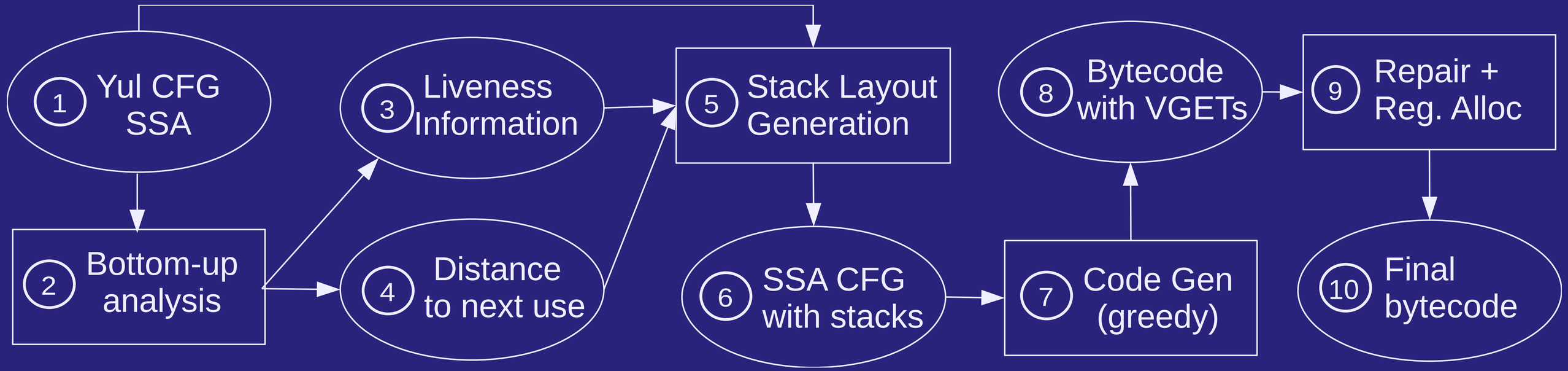
Conclusion: the memory has to be used in order to compile arbitrary EVM programs

Solidity's memory layout is compatible with using a memory region as "registers"

Assuming there are no memory-unsafe inline assembly blocks (and no recursion)



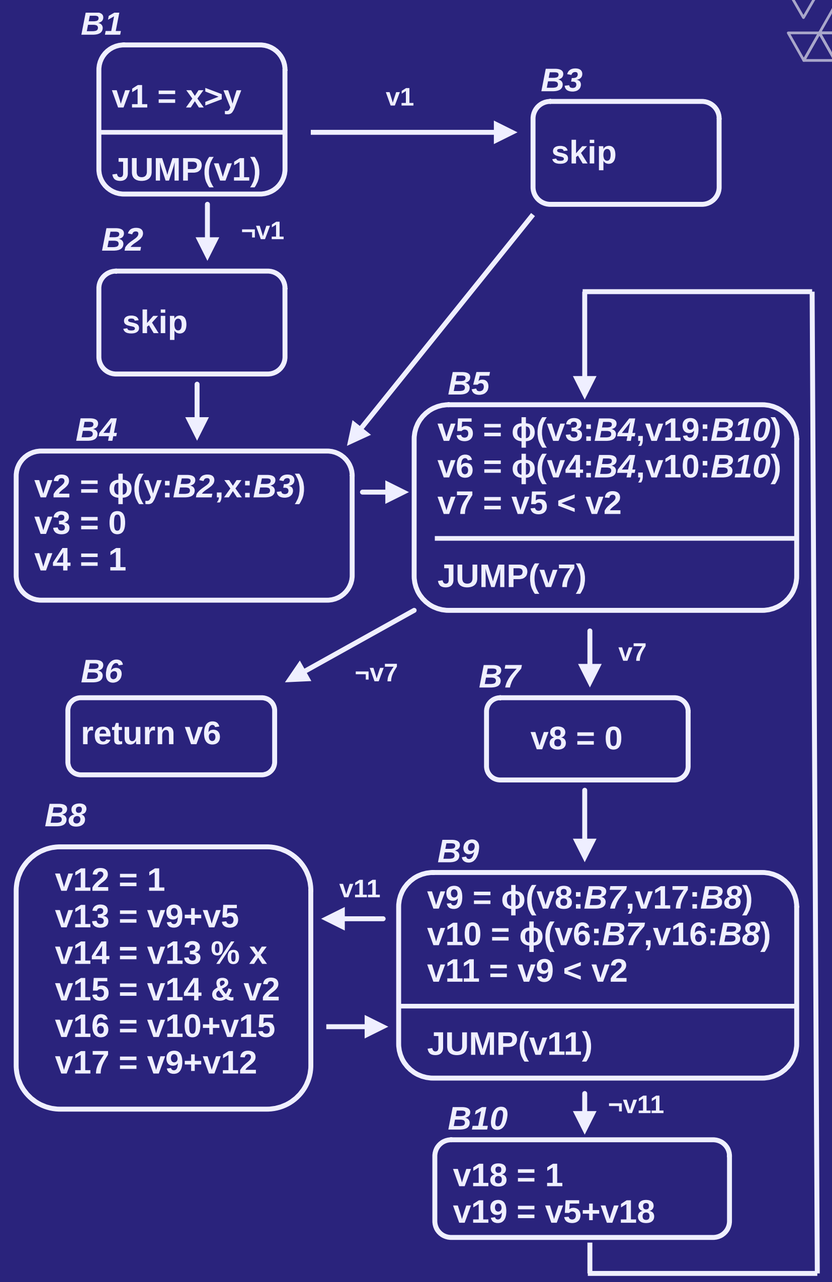
The Grey tool



In Solidity's Yul-to-EVM pipeline, liveness analysis and code generation are combined in a bottom-up pass.

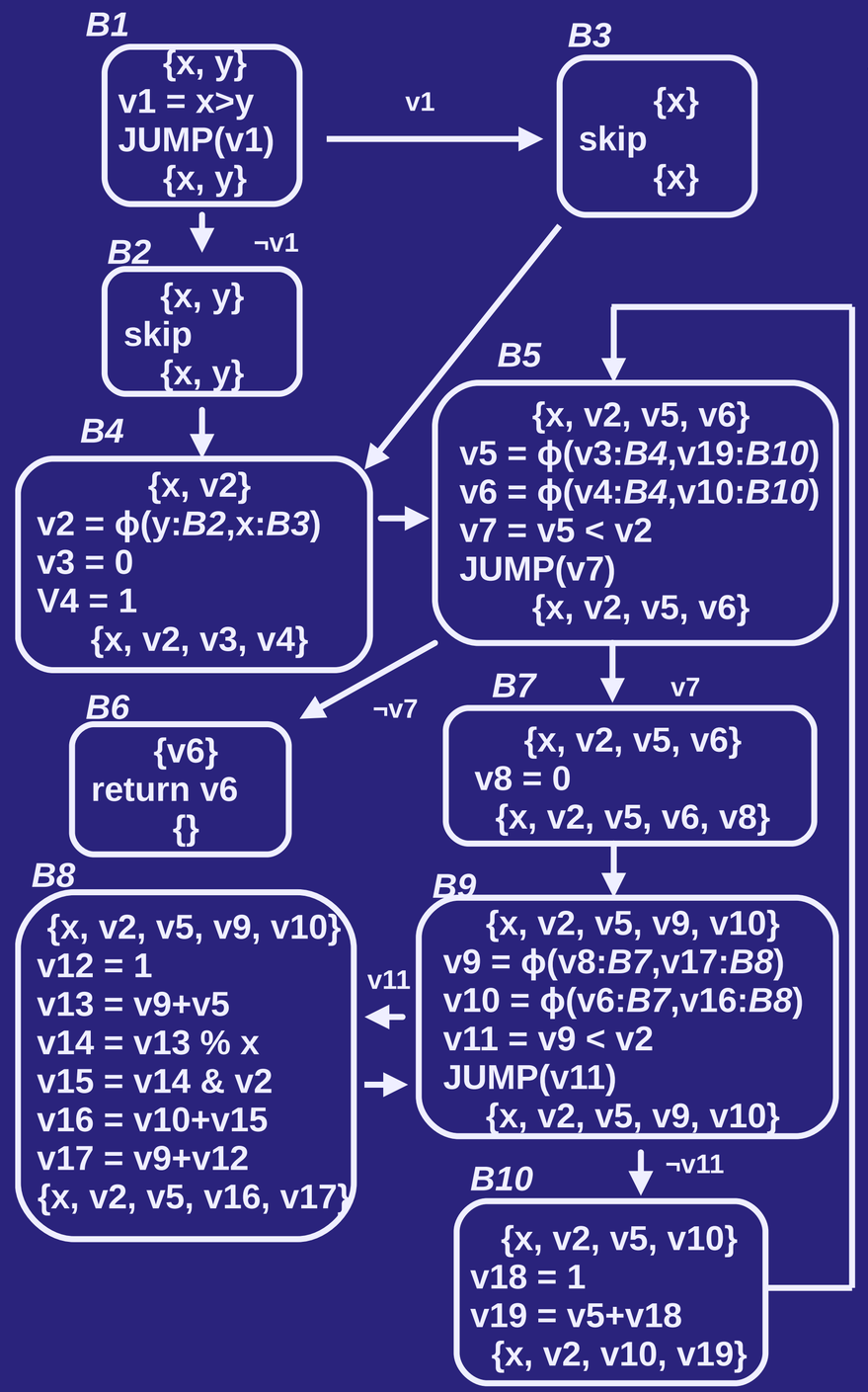
Our starting point: the Yul CFG SSA representation

```
contract SimpleYul {
    function simpleSum(uint8 y, uint8 x) public pure returns (uint256 result) {
        assembly {
            let limit := y

            // Conditional before loops
            if gt(x, y) {
                limit := x
            }

            result := 1

            // Two nested loops
            for { let i := 0 } lt(i, limit) { i := add(i, 1) } {
                for { let j := 0 } lt(j, limit) { j := add(j, 1) } {
                    result := add(result, and(mod(add(j, i), x), limit))
                }
            }
        }
    }
}
```



Bottom-up Analysis

1. The first step consists of performing a liveness analysis to determine which variables are live at the beginning and end of each block.
    a. Very efficient under SSA (two passes over the CFG).
2. We also determine the distance to the next use for each variable in each block (bottom-up information).



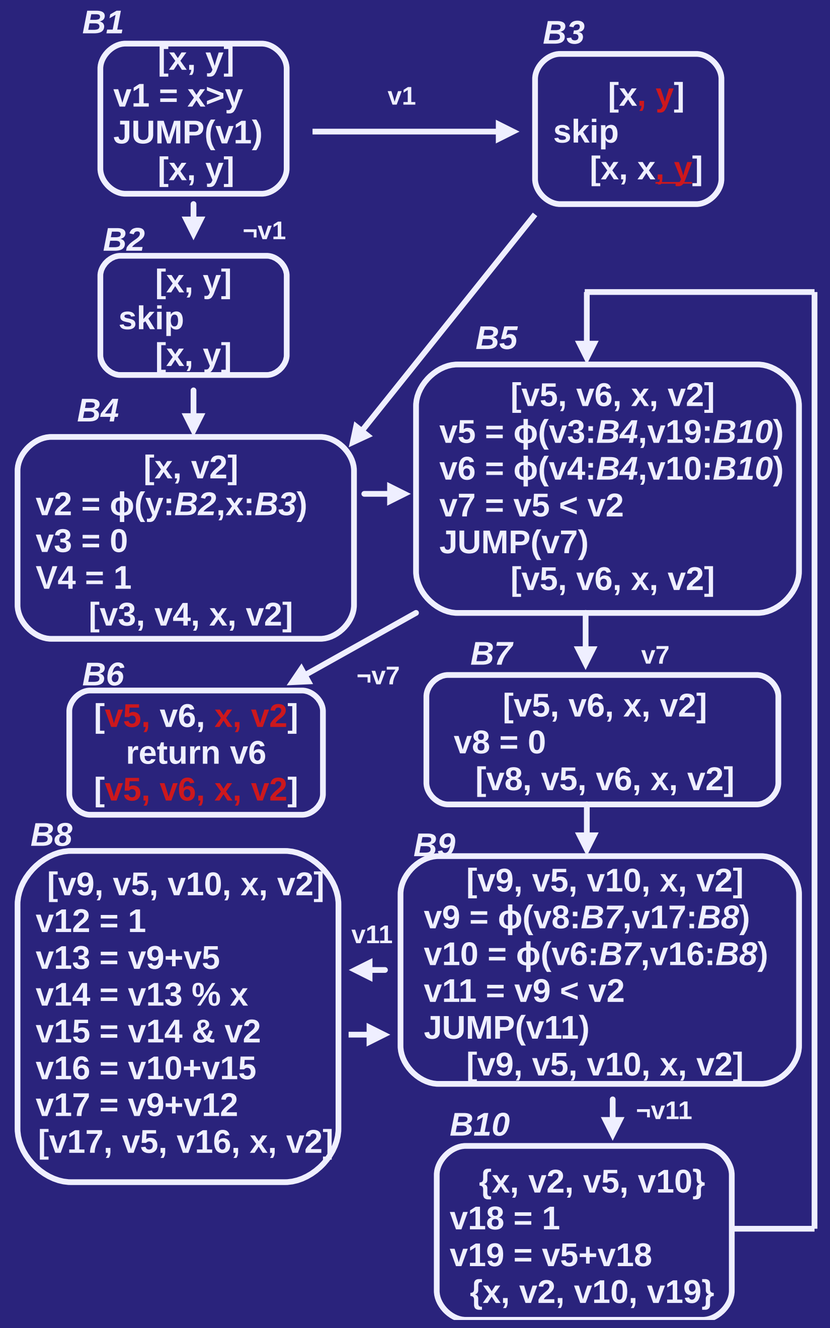
Stack Layout Generation

1. Using the liveness information, we determine the contents of the stacks before and after executing each block (i.e. propagation of variables) in a top-down traversal.

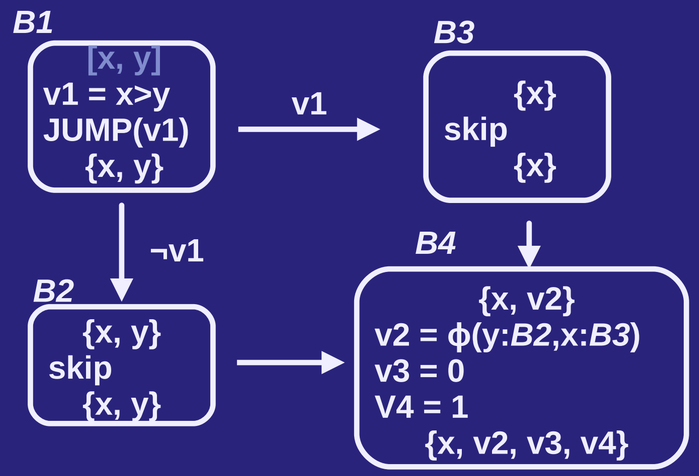
   a. Uses heuristics based on next use.

   b. We could decide to pass variables through memory at this point.

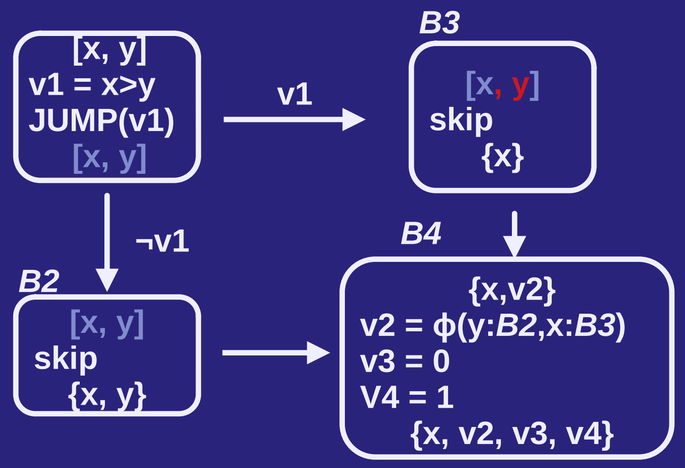
2. Garbage: crucial for avoiding too many POPs when removing variables.



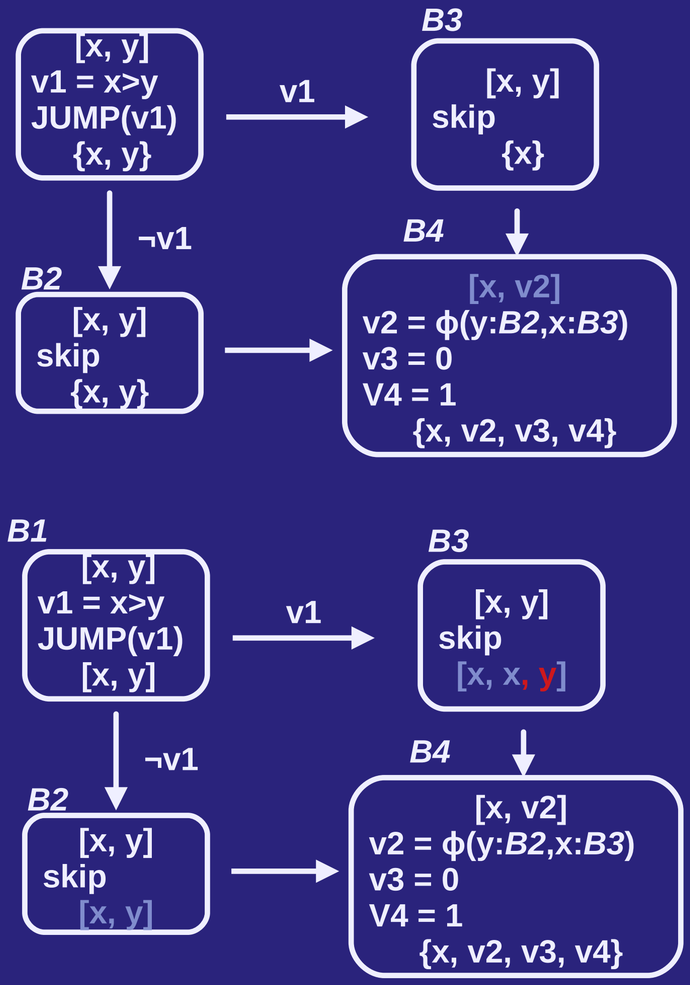
Example Stack Layouts



Initially, only B1.in-stack is generated (function args)



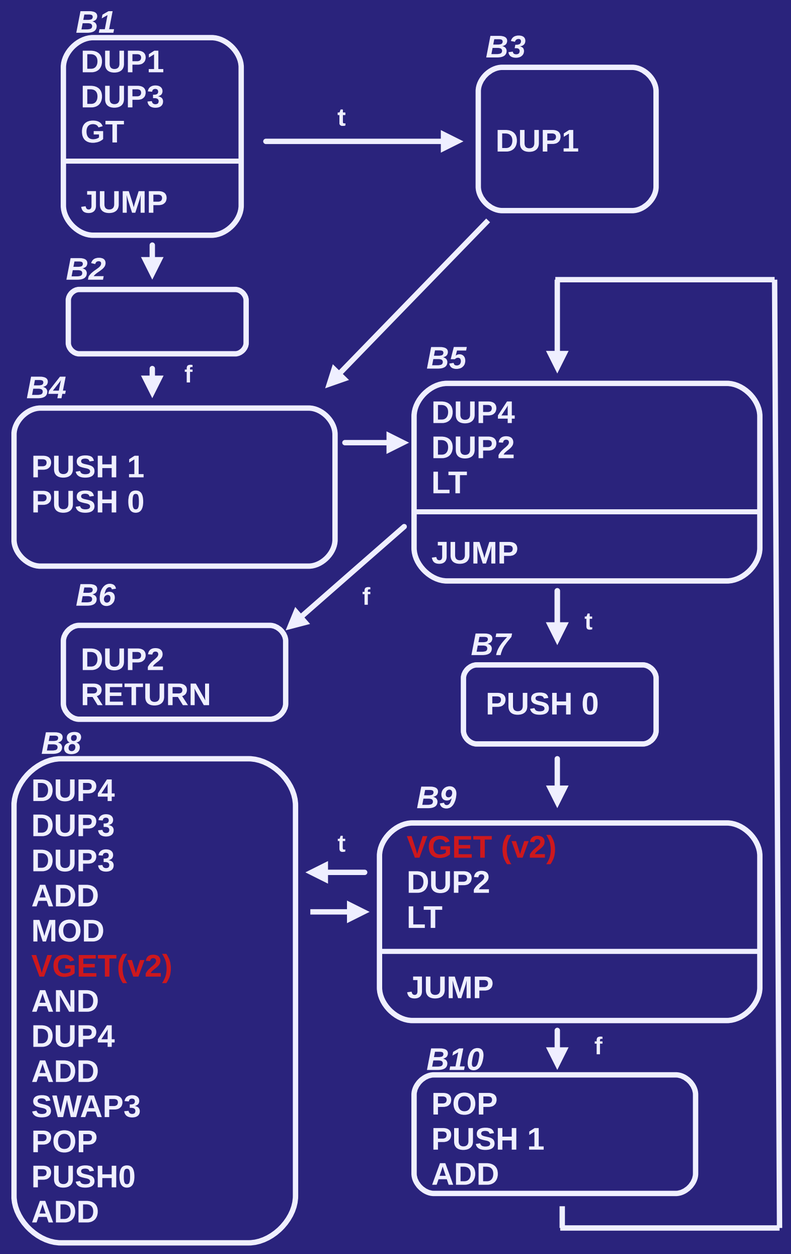
B1.out-stack is generated and propagated to B2 and B3



To generate B3.out-stack, first generate B4.in-stack and then propagate to its predecessors

Bytecode Generation (Greedy)

1. To generate EVM bytecode, we use the layout information together with the expressions to compute.
2. Our proposal: a greedy algorithm that transforms the in-stack into the out-stack and computes all expressions.
3. When a stack-too-deep access is detected, the algorithm assumes it has been stored previously in memory and marks a VGET to retrieve it



Assuming there are no DUPx/SWAPx with x > 4

Bytecode Generation – Cont.

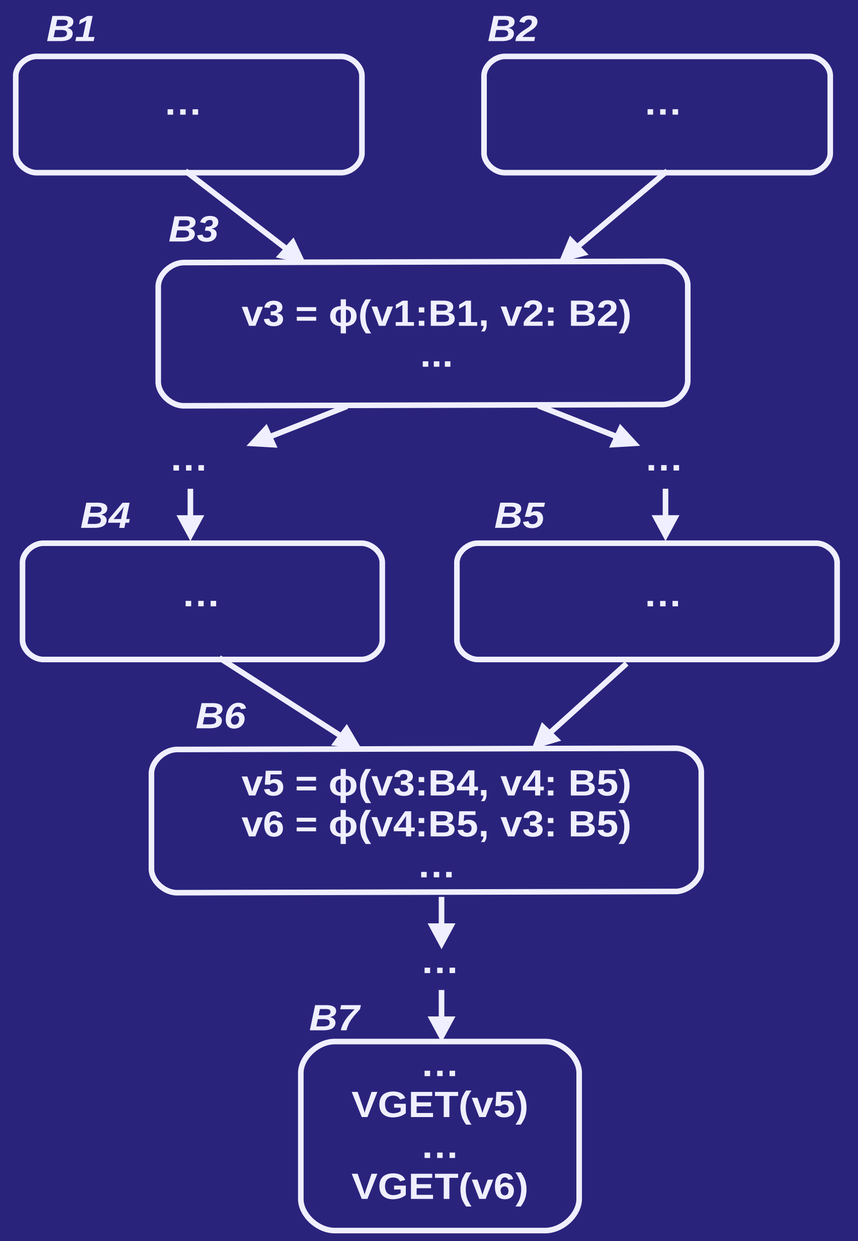
1. The greedy algorithm chooses an action that transforms the current stack closer to the target stack:
    a. Pop the topmost element if it is not used in any other computation
    b. Place a variable in their corresponding stack position w.r.t to the bottom of the stack
    c. Compute a expression still left to compute
2. When no computations can be chosen, the stack is a permutation of the target stack.

Repair unreachable accesses

1. The last step to generate executable bytecode is to replace VGET(x) with "*PUSH c MLOAD*", where c is the memory position where x is stored.
2. Key idea: our layouts respect SSA form, so we can handle accesses to memory using existing techniques that guarantee correctness.
3. Extra steps before register-allocation:
    a. Identify which phi-functions are involved (not all are needed)
    b. Determine where every variable affected is introduced in a register (not necessary where defined)

Identify affected phi-functions

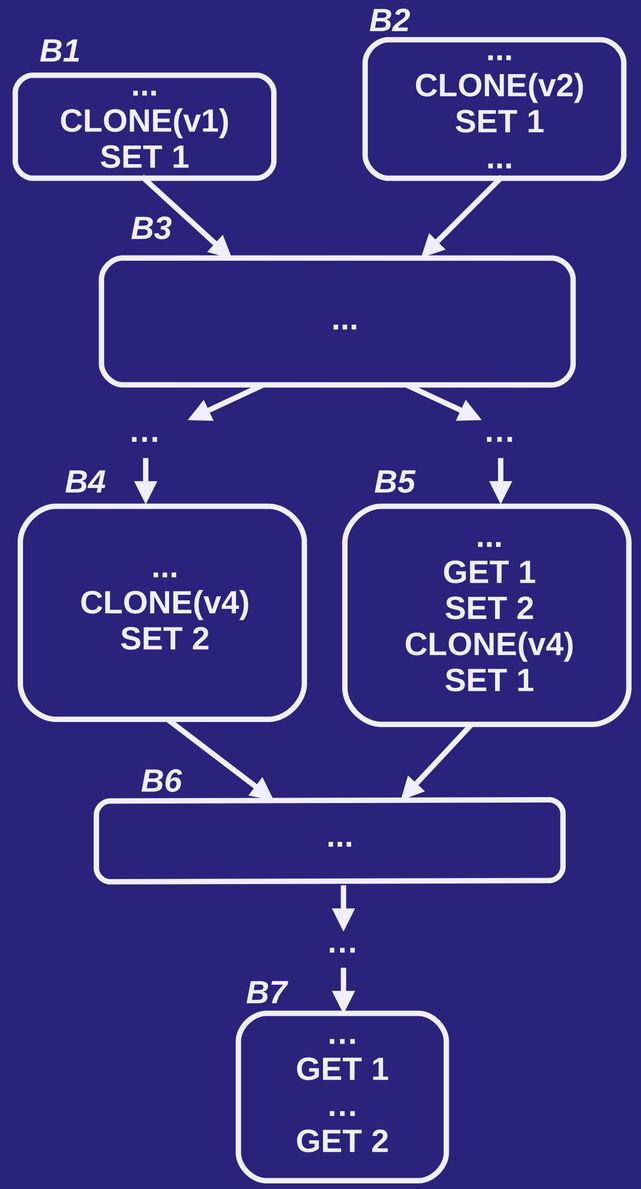
1. Every variable can be duplicated and introduced at the stack at some point.
2. Phi-functions might not follow this rule, so we need to explicitly combine their values. Eventually, all connected variables through phi-functions can be duplicated and combined.
3. We need to identify which are these phi-functions.



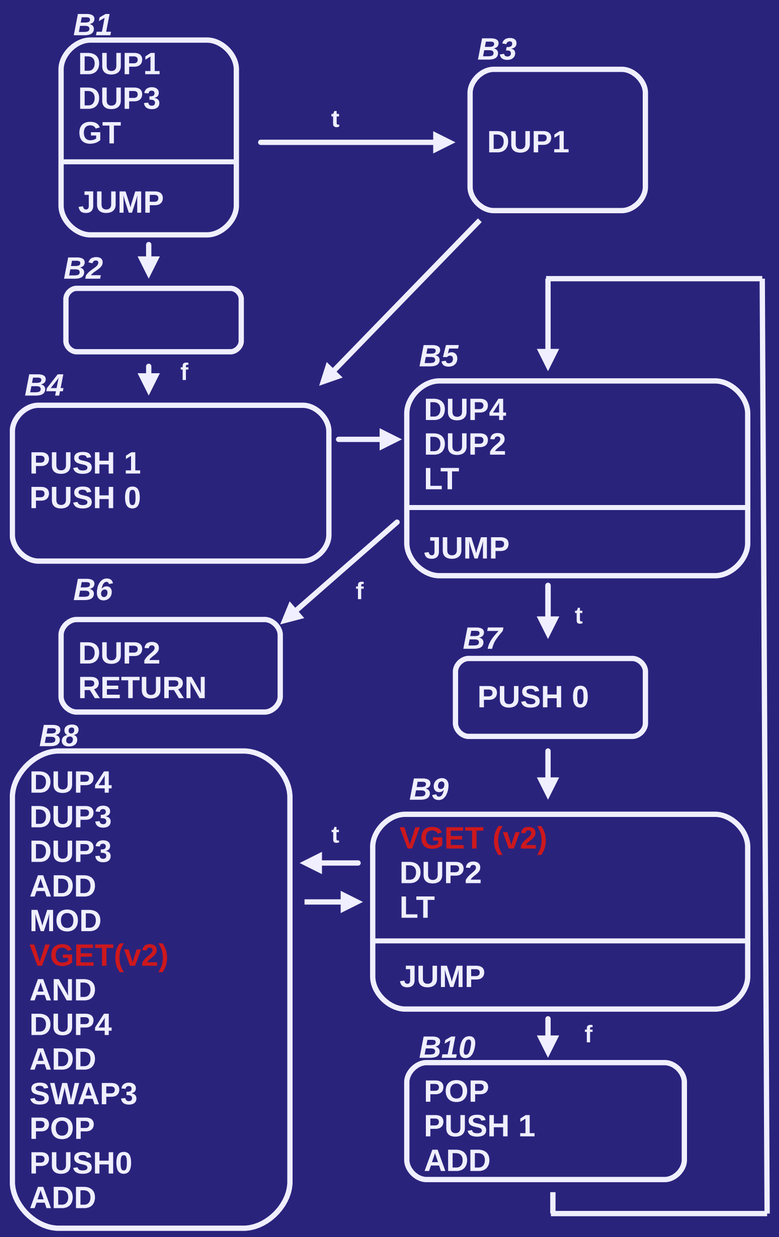
v5 and v6 cannot be duplicated, so we need to combine v3 and v4. v3 cannot be duplicated either

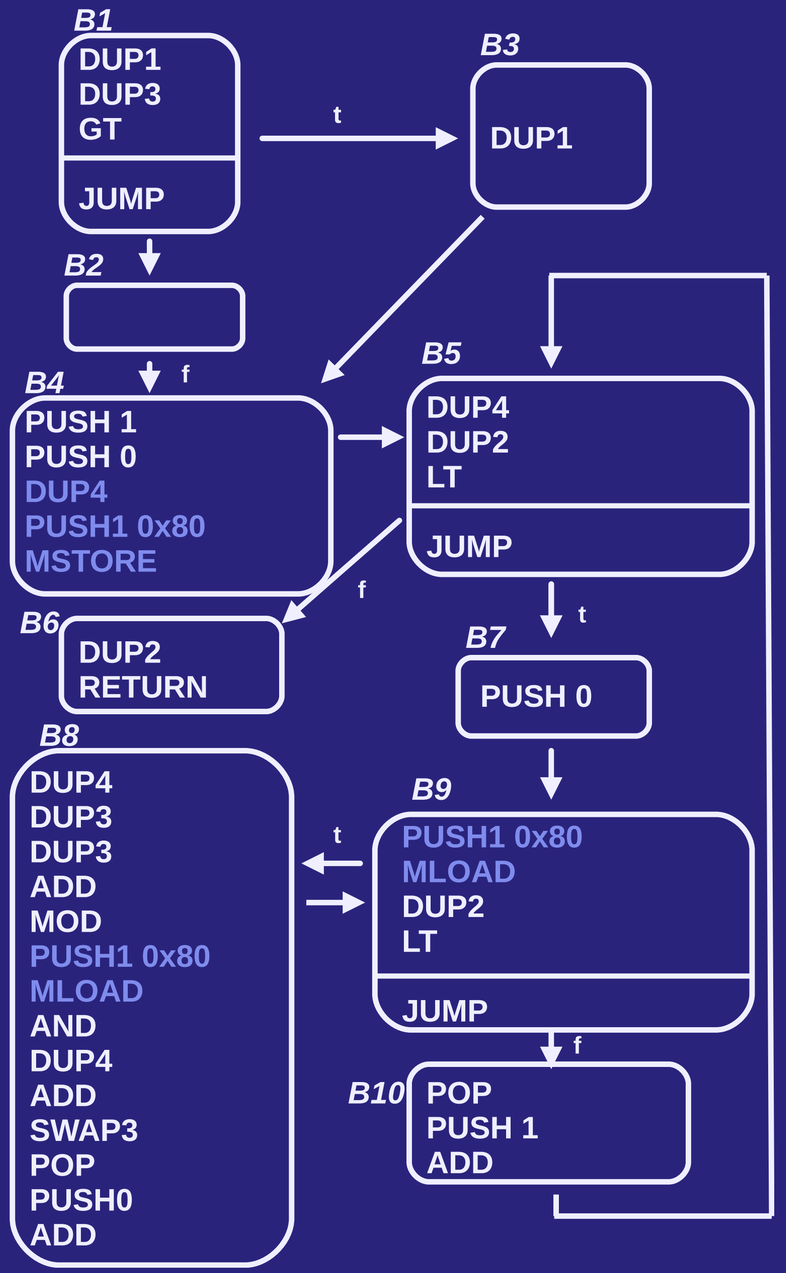
Decide when to store the variables

1. Our repair step must preserve the stack layouts, so unreachable values are propagated even if passed through registers.
2. We can store a variable the last time it was accessible in the stack before needed in a VGET.
3. Key property for efficiency: the liveness ranges of variables in SSA form correspond to subtrees of the dominance tree (a tree-like structure of the CFG) rooted in their definition



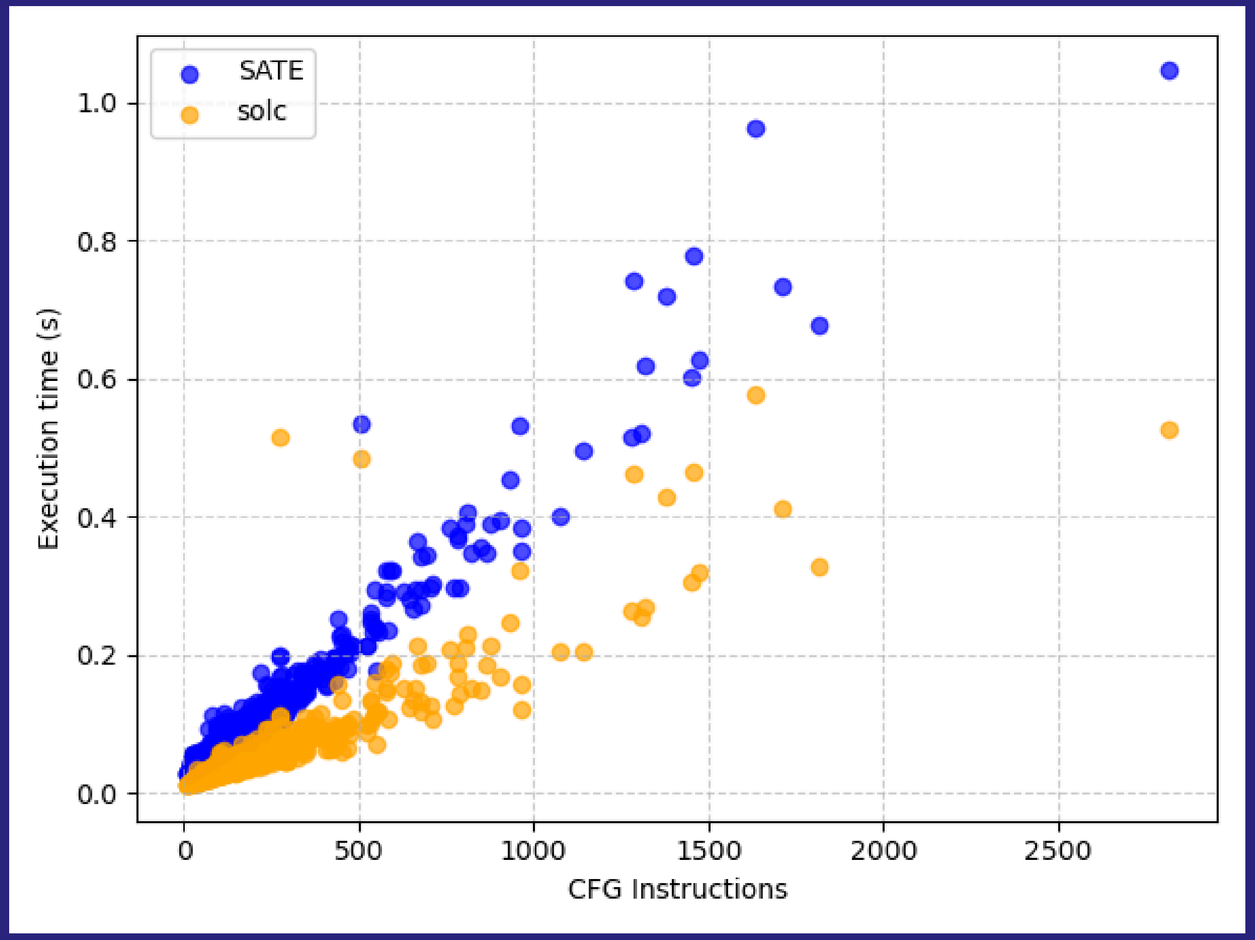
Introducing variables from phi-functions might trigger other variables to be introduced (v1, v2, v4)

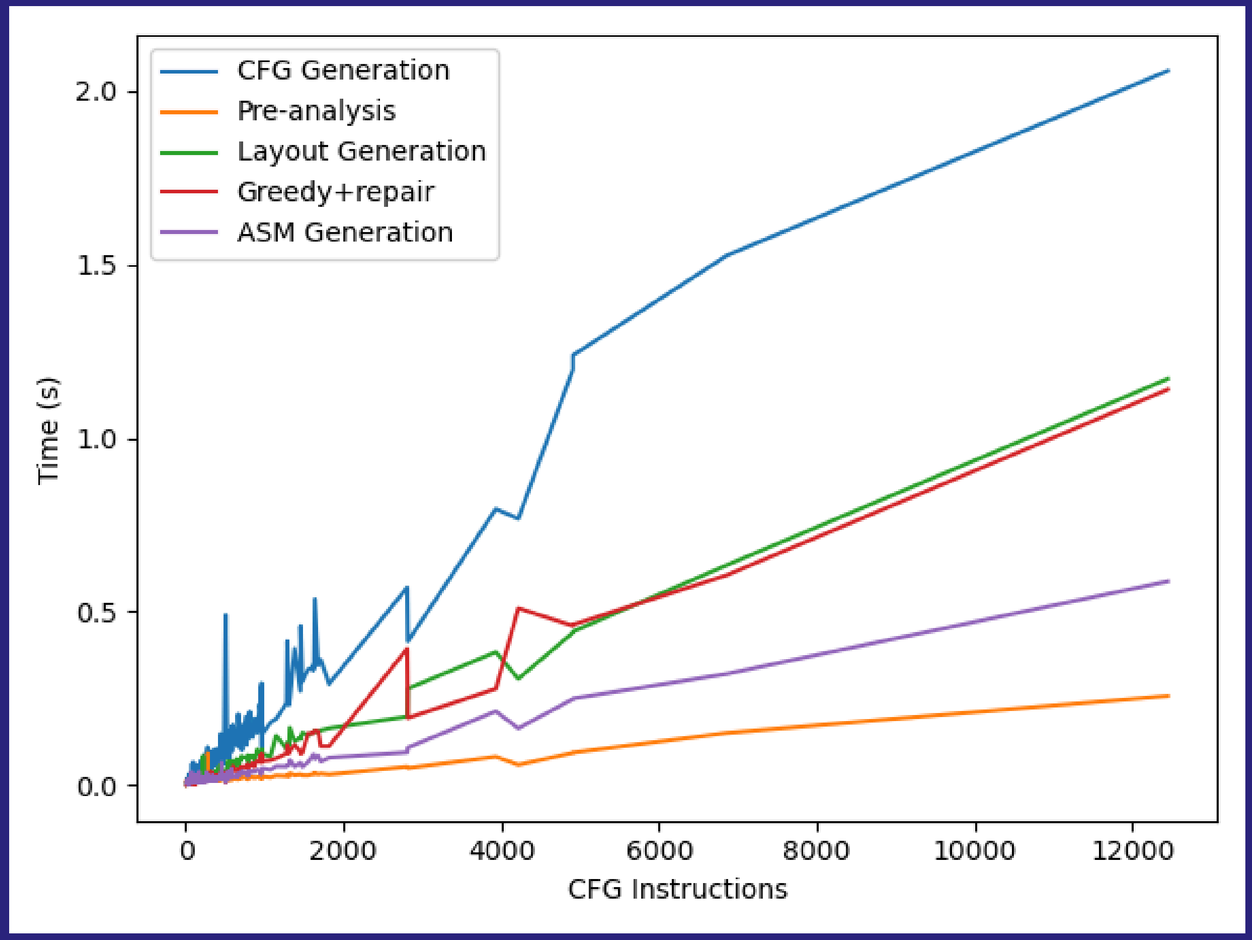

B1
DUP1
DUP3
GT
JUMP
t
B3
DUP1
B2
f
B4
PUSH 1
PUSH 0
B5
DUP4
DUP2
LT
JUMP
f
t
B6
DUP2
RETURN
B7
PUSH 0
B8
DUP4
DUP3
DUP3
ADD
MOD
VGET(v2)
AND
DUP4
ADD
SWAP3
POP
PUSH0
ADD
t
B9
VGET (v2)
DUP2
LT
JUMP
f
B10
POP
PUSH 1
ADD


B1
DUP1
DUP3
GT
JUMP
t
B3
DUP1
B2
f
B4
PUSH 1
PUSH 0
DUP4
PUSH1 0x80
MSTORE
B5
DUP4
DUP2
LT
JUMP
f
t
B6
DUP2
RETURN
B7
PUSH 0
B8
DUP4
DUP3
DUP3
ADD
MOD
PUSH1 0x80
MLOAD
AND
DUP4
ADD
SWAP3
POP
PUSH0
ADD
t
B9
PUSH1 0x80
MLOAD
DUP2
LT
JUMP
f
B10
POP
PUSH 1
ADD

Implementation & Experimental Evaluation

1. The Grey tool is implemented in Python and available online.
2. We have performed a throughout evaluation in two datasets:
    a. The Solidity benchmark set: 1402 contracts.
    b. A benchmark set of 16 contracts downloaded from Sourcify with stack-too-deep errors in the latest solc version.
3. For benchmark (a), we achieve savings of 1.9% #instrs. / 2.3% for bytes / 2.7% gas.
4. For benchmark (b), we repair all stack-too-deep cases.
5. For benchmark (a) and decreasing the max stack depth to 8, we also solve all stack-too-deep.


SATE
solc
Execution time (s)
CFG Instructions
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500


CFG Generation
Pre-analysis
Layout Generation
Greedy+repair
ASM Generation
Time (s)
CFG Instructions
0.0
0.5
1.0
1.5
2.0
0
2000
4000
6000
8000
10000
12000

Conclusions & Future Work

1. This pipeline describes how to generate EVM bytecode from the Yul CFG SSA representation, addressing the stack-too-deep for general cases.

2. This work has also served for debugging the Yul CFG SSA generation.

3. Our major limitation is that the Yul CFG SSA is missing some optimization passes (function inlining, redundant code elimination), so comparison is not direct.

4. There are several ideas we want to explore for the future:

    a. Introduce a notion of *stack pressure* and move variables direct to registers.

    b. Better heuristics for handling garbage in the layout generation phase.

    c. A better method for choosing the next computation in the greedy algorithm.

Thank you for your attention!

Q&A