Statify

November 14, 2023

Because the symbolic path traversal may lead to state explosion and too much needed user interaction which defeats the purpose of having an almost plug-n-play tool at the start of a security review. We will try to ease out a bit on symbolic execution.

The aim is to take some concrete representations:

And to approximate these concrete trajectories into an abstract one:

Coverage-guided fuzzing Stop when 100% or maximum iterations

If we meet the same JUMPDEST multiple times, write bounds. Do some BMC

Fuzzing will do the arrows in the diagram. Find entry points of the program, also find end points. End is where the arrow will start, entry is where the arrow goes.

Can we assume that the execution is complete when coverage = 100

Well if if have only visited one of those, the coverage wouldn't be 100

We could use abstract representation to replace concrete fuzzed inputs for jump conditions by a formula by checking if it fits all the trajectories.

1 CFG

When we explore the CFG, only an extremely rudimentary analysis is done. For instance, we won't resolve 'PUSH' operations when converting the bytecode to mnemonics, because this will get done later in the dynamic execution. That means that we won't even conduct any dead code analysis during that time, we just want to extract the blocks of the cfg, starting with a 'JUMPDEST' and ending either with a 'JUMP/JUMPI' or a terminating block. All of these blocks should just be stored in a 'Vec¡Block¿'.

2 Linking

The linking process aim is to create a directed graph out of the flattened cfg through fuzzing. It would be great if the fuzzing was coverage guided. For now, we can just stop once the coverage is 100

Automatized state machine breaking with ‘proptest‘ https://proptest-rs.github.io/proptest/proptest/state-machine.html

When ABI is provided, we should guide fuzzing with it. Also try to send some wrong inputs.