DATX11-23-64

Decompilation using approximate symbolic execution

Background

Sometimes, including but not limited to when playing computer security CTF competitions, one want to analyze a binary executable to understand things like:

- what the source code looked like
- what memory safety issues can be exploited with adversarial input

To do this analysis there are multiple methods that do well in different circumstances:

- Fuzzing
  - Example: AFL https://lcamtuf.coredump.cx/afl/
  - Explanation: Run the program many times with randomized input using genetic algorithms, trying to trigger as many control-flow execution paths as possible
  - Advantage: Good for branchy code
  - Disadvantage: Fails opaquely for non-branchy code
- Symbolic execution with constraint-solving
  - Example: (one component of) angr https://angr.io/
  - Explanation: Construct a symbolic expression from program input to program output, pass this to an SMT solver.
  - Advantage: Complete description of "simple" code
  - Disadvantage: Fails opaquely for complicated code, state explosion for branchy code
- Decompilation to pseudo-C
  - Example: Ghidra https://ghidra-sre.org/
  - Explanation: Re-construct control-flow and functions, then present pseudo-C in an interactive editor allowing e.g. variable renaming.
  - Advantage: Can be explored interactively
  - Disadvantage: Decompiles incorrectly when encountering unknown tricks, implicit function state, single functions cannot be fuzzed/symbolically evaluated

I believe a better decompiler should use symbolic evaluation as a way to decomposing the analyzed program into smaller parts.

Project description

Write a program that

1. loads an executable binary, disassembles it and lifts it to a platform-independent representation (use existing tools)
2. recovers its control-flow graph in a format suitable for decomposition into pure functions (i.e. tracking what code can access what part of memory) (most likely by writing novel code)
3. does at least one of
   (a) presents the control-flow graph of functions to the user in a typical interactive decompiler manner, including information about their syscalls, image and other properties relevant for big-picture program analysis
   (b) allows SMT solving and/or emulated fuzzing of single functions

The overall goal is to bridge the gap between automated program analysis such as fuzzing and SMT solving, which do very well on small simple cases but often fail opaquely in the large, and typical decompilation which allows an interactive reverse engineering process but leaves a lot of manual work on small components that could be done automatically if the decompiler could do guaranteed-correct control-flow component decomposition.

This could possibly be built on top of angr, but manipulating angr's symbolic representation of code might be too cumbersome. In that case the angr components CLE (executable loading) and PyVEX (lifting to Valgrind IR) could be used, with a home-rolled symbolic emulator.

Suggested reading material

- https://docs.angr.io/
- https://github.com/angr/cle
- https://github.com/angr/pyvex
- ask me

Other

Prerequisites

Knowledge/experience of some of

- Algorithms and data structures (persistent data structures might be useful to store many program memory snapshots at different points in the execution)
- CTF or reverse engineering
- Assembly language (since the project can be though of as implementing an emulator)
- Computer algebra (we must in particular (somewhat simplified) find a hueristic for when a symbolic expression is sufficiently similar to a previous expression that we can skip reprocessing it)
- ELF loading, how dynamic libraries work

Targeted students

D, DV, IT, TM

The final report will be written in Swedish

Proposed by

Loke Gustafsson (Y3 TM)