Automated Verification of RISC-V Kernel Code

Antoine Kaufmann

Big Picture

- Micro/exokernels can be viewed as event-driven
  - Initialize, enter application, get interrupt/syscall, repeat
- Interrupt/syscall handlers are bounded
- Most of the code fiddles with low-level details in some way
  - -> messy, high level language does not help

Idea:

- Use SMT solver on instructions and spec
  - (And see how far we get)

Overview

1. RISC-V Z3 model

2. Kernel + Proof

RISC-V

- Free modular ISA from Berkeley
- Clean slate, compact, and no legacy features
  - 100 pages of spec for user instructions (including extensions)
  - 60 pages for kernel features
- 62 Core RV64-I instructions
  - Basic register operations, branches, linear arithmetic, bit ops
- Supports full blown virtual memory, or base+bounds

RISC-V SMT Model Status

- Only 8 RV64-I instructions still missing:
  - fence(i), rdcycle(h), rdtime(h), rdinstrret(h)
- Supported kernel features:
  - Transfers between protection levels: syscall/traps
  - Base and bounds virtual memory
- Missing kernel features:
  - Modelling interrupt causes
  - More than 2 privilege levels
  - Full page table based virtual memory
- Runs (and passes) provided riscv-tests for instructions
  - -> Demo

Model: The first attempt

- Pure Z3 expressions for fetch and exec of instructions
  - `fetch_and_exec :: machine state -> machine state`
- Having pure Z3 expressions everywhere is convenient
- Use `simplify` after every step to keep compact
- But: Non-trivial Conditional branches cause blow-up
  - Simplify won't be able to cut down much in next step
  - Expressions built bottom up, lot's of unneeded work

Model: “split” state

- At conditional, check which branches are reachable
  - Only build up reachable branches
- Keep branches separate (while storing path condition)
  - Further process each expression separately
- Expressions stay smaller and are easily simplified
- The Kernel code is not very “branchy” so number is small

- But: Python implementation gets messy
  - Basically monads -> manually building continuations

Kernel: Spec

- Given a valid system config kernel will initialize internal state correctly and enter the first application
    - System config: Applications to load (entry, base, bound)
- If we get a yield system call: switch to other application
    - Kernel state still contains this application's state
    - New application's state is restored correctly
- A sbrk() system call will increase the bound if possible while maintaining isolation
- If application faults/calls invalid syscall it is terminated

Kernel: Initialization

- Run initialization code with abstract config until it reaches userspace
  - Ends up with separate expressions depending on number of applications (1-8 currently)
- Require: Valid Config
  - Non-overlapping applications, entry PC in bounds and aligned
- Ensure: First app running and internal state correct
  - Read-only parts not modified
  - Base-bounds VM enabled
  - PC, mbase, mbound set to first app's values
  - Current process points to first process handle
  - Valid apps in config marked as valid in internal state

Kernel: Induction (Proof: WIP)

- Run event handlers (system calls, faults), and show invariants preserved and event handled correctly
- Start in symbolic state
  - Load only read-only sections of kernel ELF
- Require: Valid kernel state
  - Current app valid and it's base and bound set
  - Application's base and bounds are non-overlapping
- Ensure: Event handled correctly and valid kernel state
  - Read-only parts of kernel not modified
  - Handle event correctly (yield, fault, sbrk)
  - Current app valid and it's base and bound set
  - Application's base and bounds are non-overlapping

Future Work

- Finish kernel proof for base-and-bounds VM
- Add additional system calls (e.g. starting new process)
- Model page table based VM
- Speed up verification
  - Most of the time is passed in `z3.simplify()`
  - Could parallelize when splitting state