Design of a Symbolically Executable Embedded Hypervisor

Jan Nordholz <j.nordholz@tu-berlin.de>

15th EUROSYS, 27-30 April, 2020

full presentation

PHIDIAS – Type I Embedded Hypervisor

- Design paradigm: fully embrace static use cases – no compromises
  - No creation/destruction of VMs
  - No scheduling
  - No memory allocation/reclamation
  - No IRQ registration, rerouting, …
  - No migration of VCPUs across physical cores
    → hypervisor executes independently on each core
  - No dynamic creation of inter-VM communication channels

- Use case examples:
  - Control units in automotive IT
  - Measuring instruments subject to metrological certification

Offline Configuration Toolkit

- Necessary functionality moved into offline configuration toolkit:
  - Reads in system configuration (XML)
    - Target architecture and SoC
    - Number of VMs, memory requirements, desired IPC channels, IRQ pass-throughs…
  - Drives compilation of hypervisor
    - Selects subset of compiled modules
    - Guesses / probes for cross-compiler
  - Assigns physical memory resources, allocates virtual address ranges
  - Generates all page tables (stage-1 per HV instance, stage-2 per VM)
  - Builds schedule
  - Emits tree of C structs describing desired system objects
    - VCPUs, scheduler configuration, IRQ handler table, …

    → compiled and (loosely) linked to hypervisor image
  - Wraps final hypervisor image into necessary boot clothing (e.g., uimage)

Remaining Hypervisor Runtime Functionality

- What modules are still there at runtime?
  - Scheduler? No. VM Dispatcher (context switch / state save+restore)? Yes.
  - MM Subsystem? No, only setting of nested paging controls.
  - IRQ Handling? Yes, using a fixed dispatch table.
  - Device Drivers? Yes, bare minimum (IRQ controller, timer, CPU virt. ext.).
  - Device Emulation? Partially:
    - Devices tied into HW virtualization: yes (usually IRQ controller and timer).
    - Other devices? No, but PHIDIAS supports reflection of nested faults into another VM.
  - Inter-VCPU Communication? Yes:
    - Among VCPUs of a single VM: through virtual IRQ controller (virtual IPI emulation).
    - Across VMs: “virtual IRQ” capability allows one VM to trigger another.
  - Trap / Fault Handler? Yes:
    - Architectural traps, faults on emulated MMIO ranges: yes.
    - Hypercalls: only for triggering vIRQs and for reflection management.

Symbolic Execution Framework

- Implication of our design: all system objects are known a priori
  - Number (and memory location) of VMs, vIRQ lines etc. fixed at compile time

  → very limited state space of hypervisor

- (Recap) Common OS proof approach: abstraction and refinement
  - (usually) source code ⇔ abstract specification
  - Allows reasoning to capture abstract properties such as “correctness”
  - Very labor-intensive (e.g. interactive theorem proving)
  - Result is generic (does not depend on concrete instantiation)

Symbolic Execution Framework

- Implication of our design: all system objects are known a priori
  - Number (and memory location) of VMs, vIRQ lines etc. fixed at compile time

  → very limited state space of hypervisor

- Unique proof approach for PHIDIAS: directly analyze machine code
  → symbolic execution
  - Machine code ⇔ intermediate invariants:
    - No deadlocks
    - Suspending/resuming VCPUs is performed correctly
    - Data structures of hypervisor are kept sane
  - Checking for “correctness” property would require abstract specification
  - Automated (“push-button”) analysis
  - Result is bound to a specific instance (i.e. compiled image)

Symbolic Execution Framework

- Symbolic Execution: commonly used to analyze userspace binaries
  - ISA support usually only covers unprivileged subset
- Adoption of established framework would require adaptation:
  - Addition of privileged instructions
  - Addition of privileged resources (e.g., control registers)
  - Special handling of privileged operations
    - Many of those would require aborting the current execution trace:
      - Changing core system controls (paging on/off, cache on/off, access bits on/off, …)
      - Modification of the current address space
- Alternative: custom solution, purpose-built for executing our HV
  - Drawback: recognizes minimal set of instructions; ARMv8 only

Current State of Development

- Supported Architectures: ARMv8-A, ARMv7-A, MIPS (VZ), x86_64
- Supported SoCs: RK3399, HiKey 2, RPi 3, RPi 2, Cubieboard, Qemu virt
- Proof Engine: ARMv8 only
- Push-Button Verification Times: scales with #VCPUs, <8 VCPUs ⟶ <2h

- Overall Implementation Effort
  - ≈11 kLOC HV (C + Assembler), ≈4.5 kLOC used per instantiation
  - ≈6 kLOC configuration toolkit (C)
  - ≈7.5 kLOC proof engine (C), using Z3 as SMT backend

Current State of Development

- Being worked on:
  - RISC-V support (HiFive1 rev B)
  - Transition from self-written to a mature symbolic execution framework
  - Extension of prover results towards abstract properties
  - Release as open source project

- Aspects worth investigating:
  - Analyze / optimize cache and TLB footprint of HV code paths
    - Tune / rewrite bootable HV image to reduce footprint
    - Measure / improve worst-case latency of hot paths (IRQ delivery, frequent traps)
  - Try reintroducing dynamic aspects under our umbrella of „pure staticness“
    - HV-based big.LITTLE core switching
    - Pseudo-Ballooning by switching between multiple pregenerated sets of page tables
    - Shadow paging

Thank you for watching!