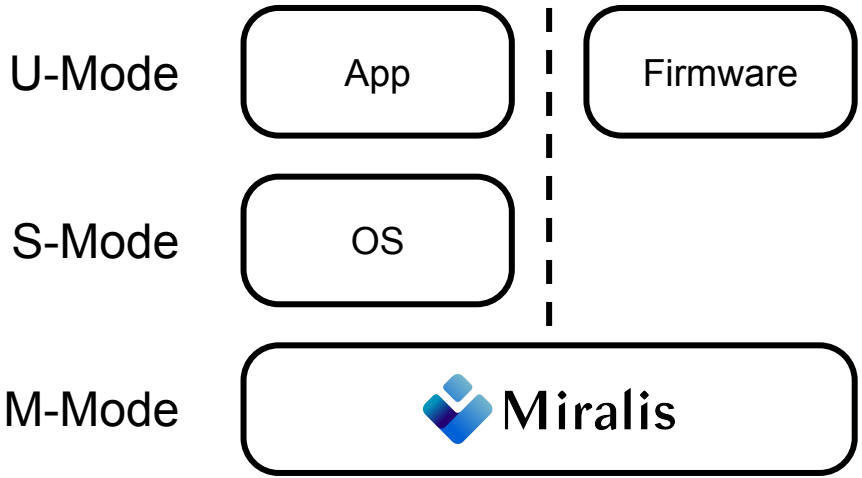
Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet

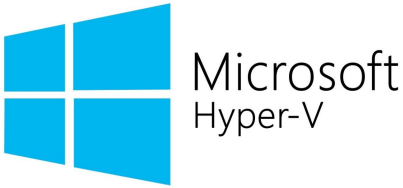
Charly Castes[1] François Costa[2] Nate Foster[3]
Thomas Bourgeat[1] Edouard Bugnion[1]

HotOS 2025

Cornell University

Hypervisors are Critical Infrastructure





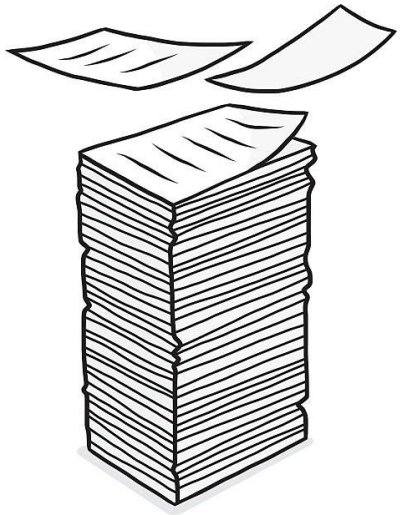
Hypervisors are Hard to Build

A hypervisor has two responsibility:

- *Configure* hardware
- *Emulate* hardware

Hypervisors are Hard to Build

It is hard, because hardware is complex



5057 pages

3347 pages

842 pages

14568 pages

Formal Verification is Hard

We would like hypervisors to be verified

It is hard because:

- Writing a spec is hard
- Writing (and maintaining) a proof is hard

Lightweight Hypervisor Verification

Can we automate the verification of a hypervisor?

Can we skip writing the spec and the proof?

Lightweight Hypervisor Verification

Yes, we can!

- Hypervisors are very structured systems
- Executable ISA specs are becoming the new norm

RISC-V

arm

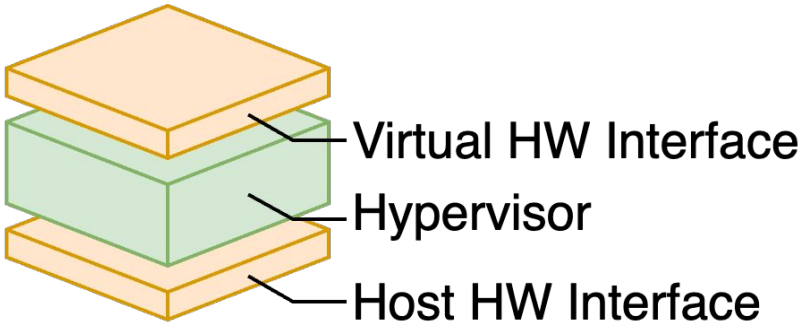
Overview

1. Hypervisor Correctness
2. Faithful Emulation
3. Faithful Execution
4. Verifying Miralis

Hypervisor Correctness

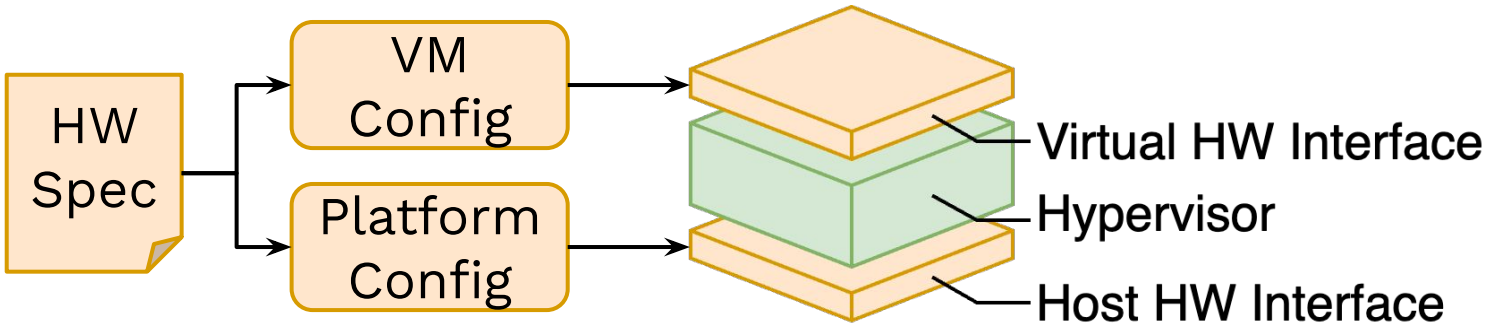
The Hardware Burger

An hypervisor exposes a virtual hardware interface



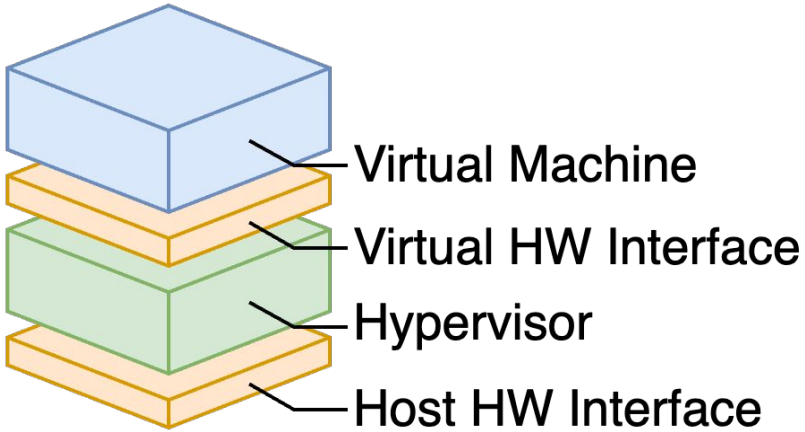
The Hardware Burger

An hypervisor exposes a virtual hardware interface



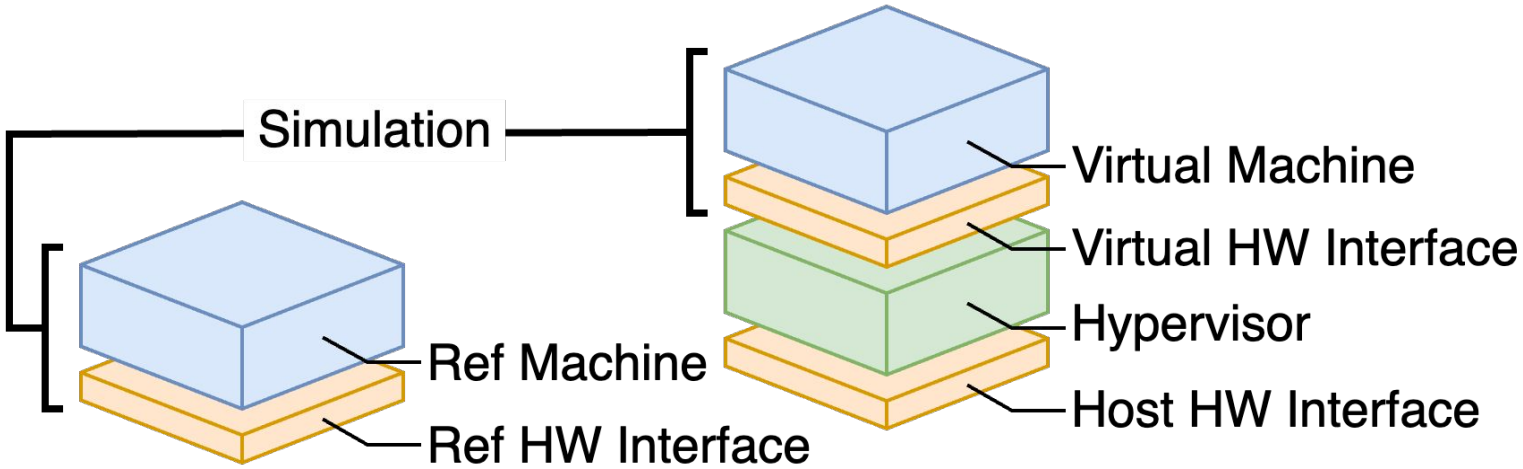
Hypervisor Correctness

A VM must execute as it would on a reference machine



Hypervisor Correctness

A VM must execute as it would on a reference machine



The VM must be a simulation of a reference machine

Trap & Emulate Hypervisors

Formal requirements for virtualizable third generation architectures

1974, GJ Popek, RP Goldberg

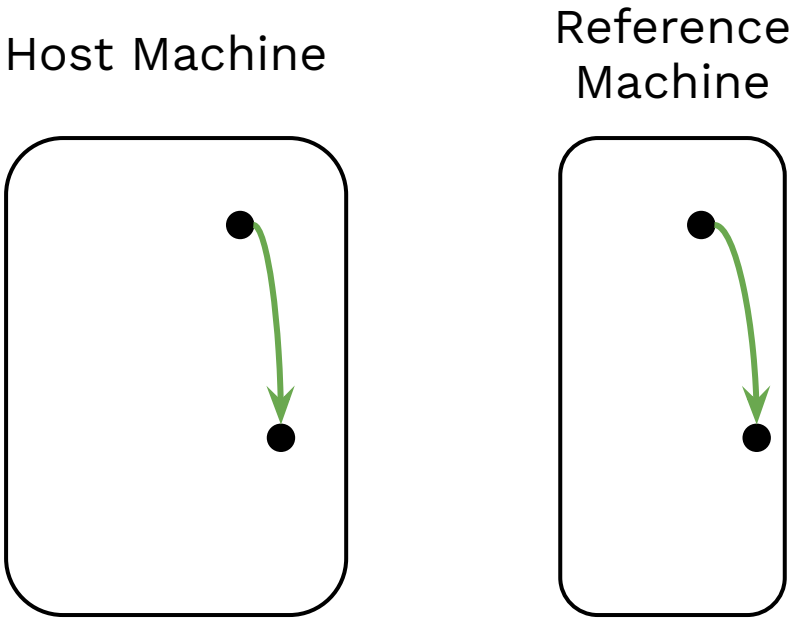
Modern architectures are *virtualizable*

↳ Hypervisors rely on *trap & emulate*

Trap & Emulate Hypervisors

Two kinds of instructions:

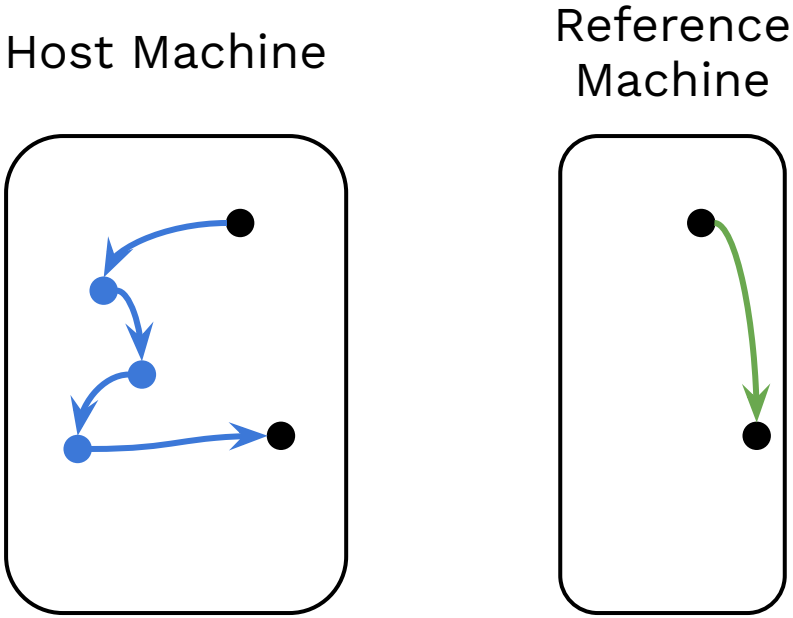
- **Unprivileged**: executed directly in hardware
- **Privileged**: trap to the hypervisor



Trap & Emulate Hypervisors

Two kinds of instructions:

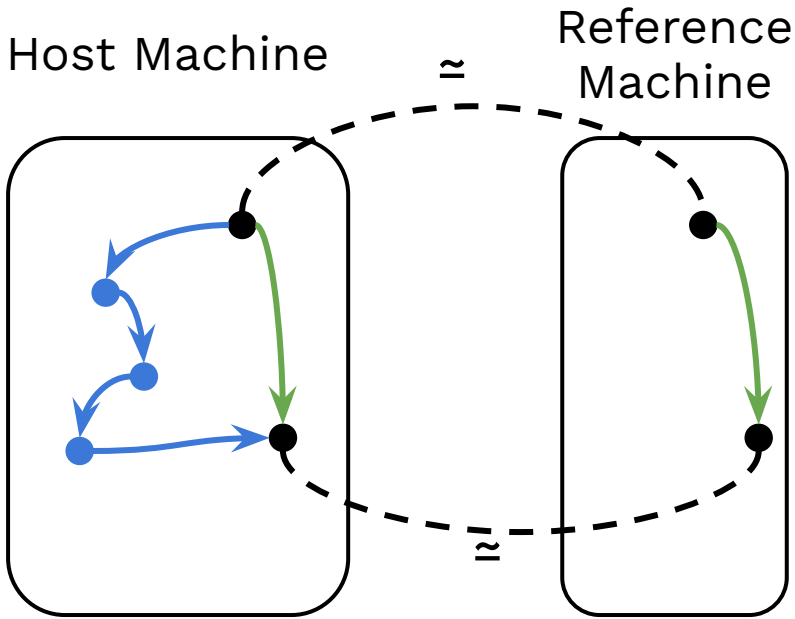
- **Unprivileged**: executed directly in hardware
- **Privileged**: trap to the hypervisor



Trap & Emulate Hypervisors

Two kinds of instructions:

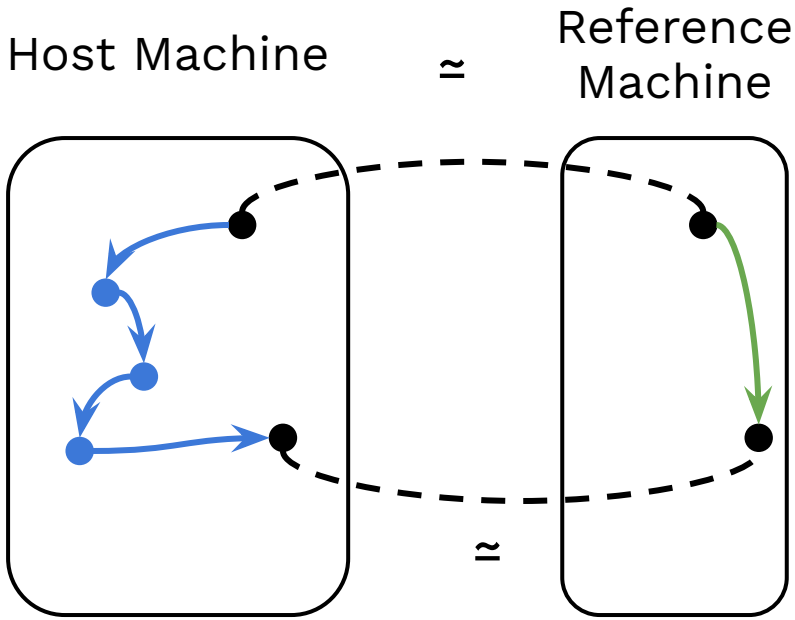
- **Unprivileged**: executed directly in hardware
- **Privileged**: trap to the hypervisor



Faithful Emulation

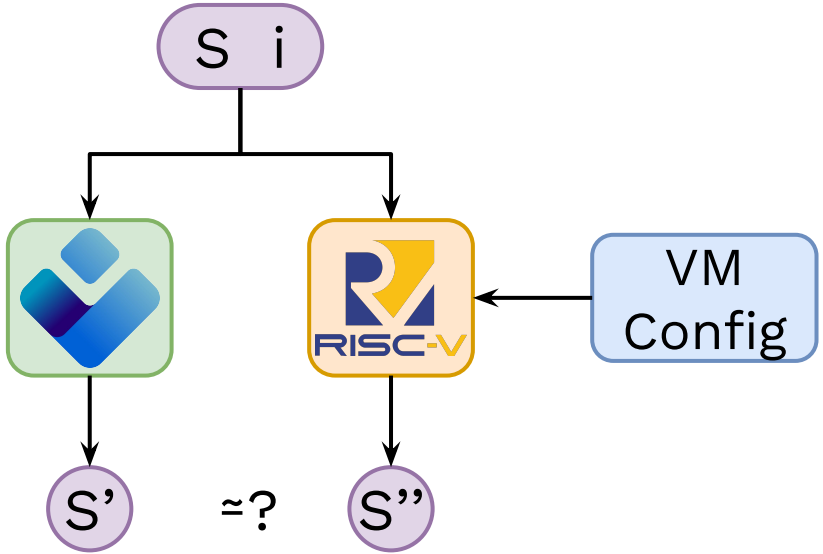
Faithful Emulation

The hypervisor should accurately emulate privileged instructions



Faithful Emulation

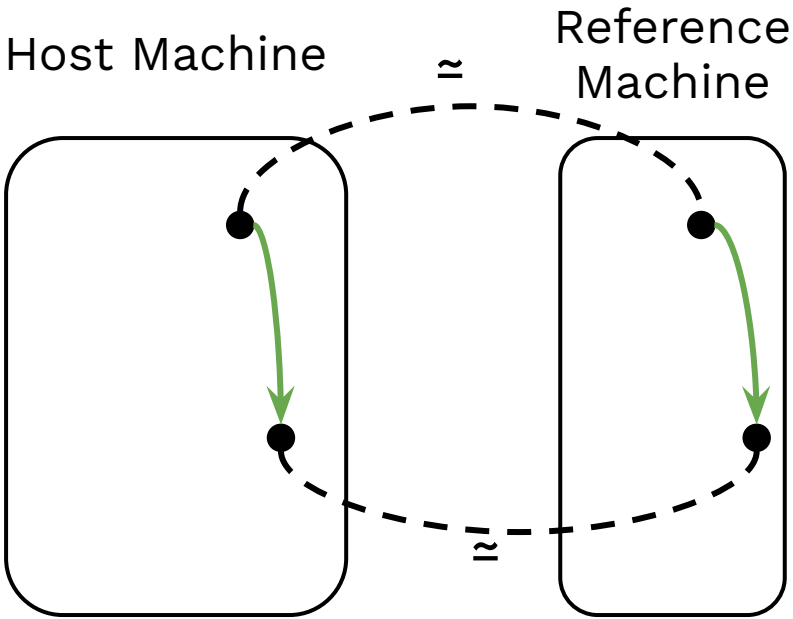
For all input state and privileged instruction, the hypervisor produces the same state as the reference machine



Faithful Execution

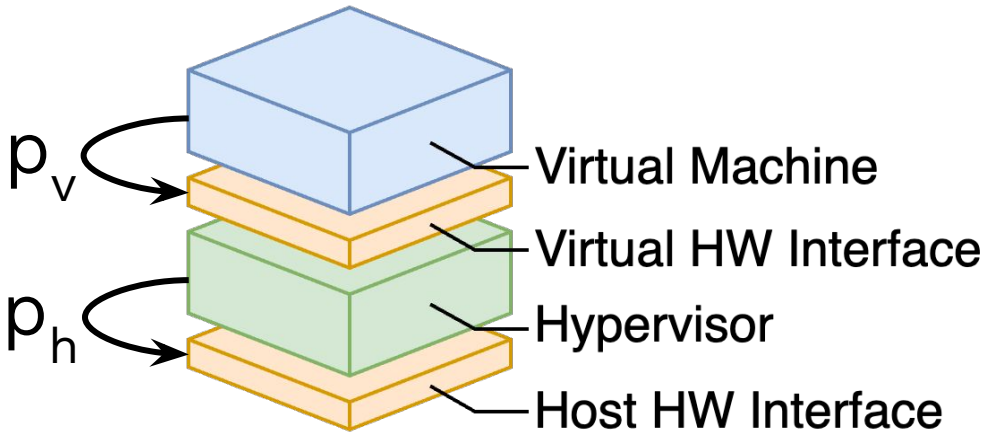
Faithful Execution

Direct execution should be indistinguishable from a reference machine



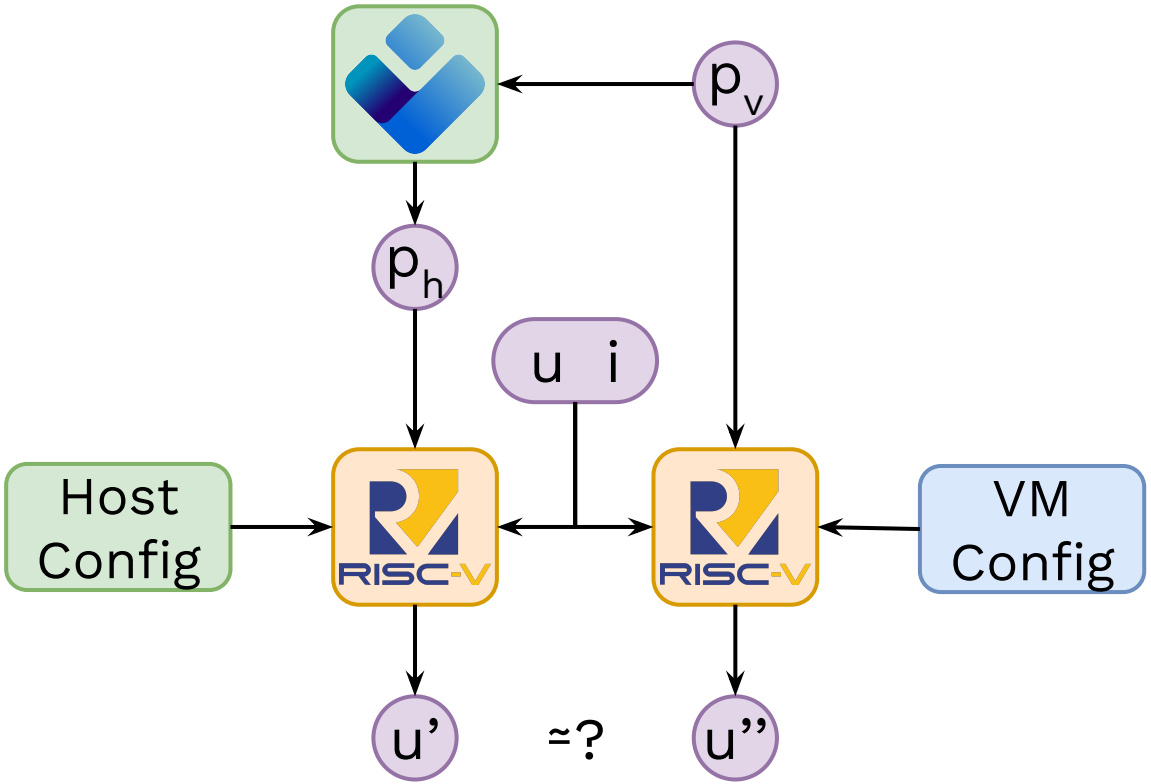
Faithful Execution

The behavior of instructions depends on the privileged state



Faithful Execution

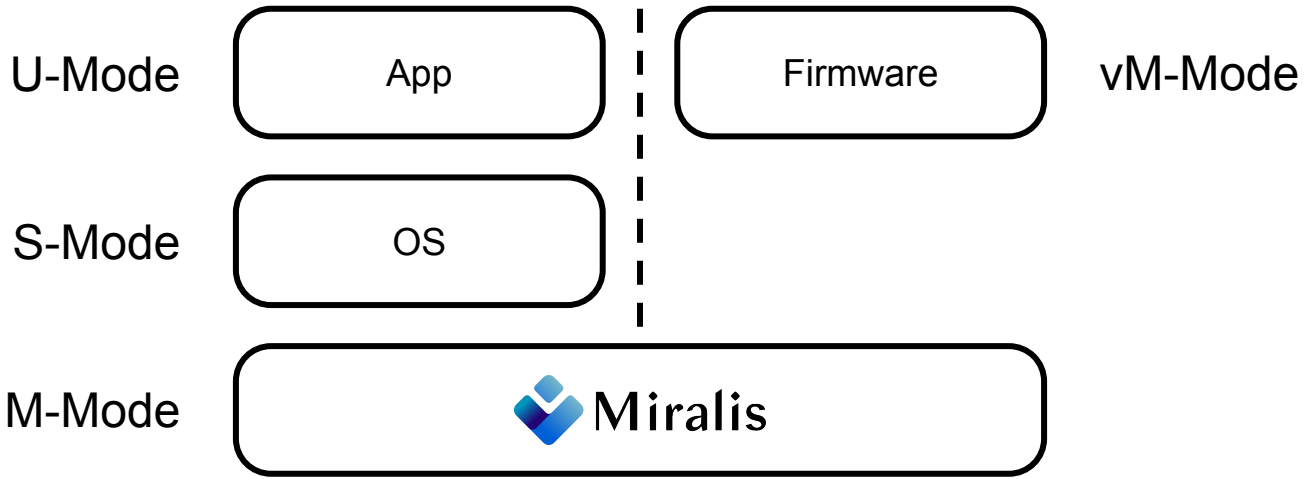
The host hardware must be programmed to execute as if the VM was running on the reference machine



Verifying Miralis

Miralis Overview

Miralis is a RISC-V virtual firmware monitor



Miralis Overview

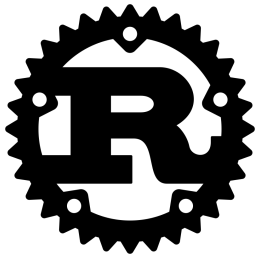
Pain points during development:

- Emulation of 84 privileged registers
- Virtual interrupts losses
- Memory isolation

} Now all verified!

Verifying Miralis - A Rust RISC-V Model

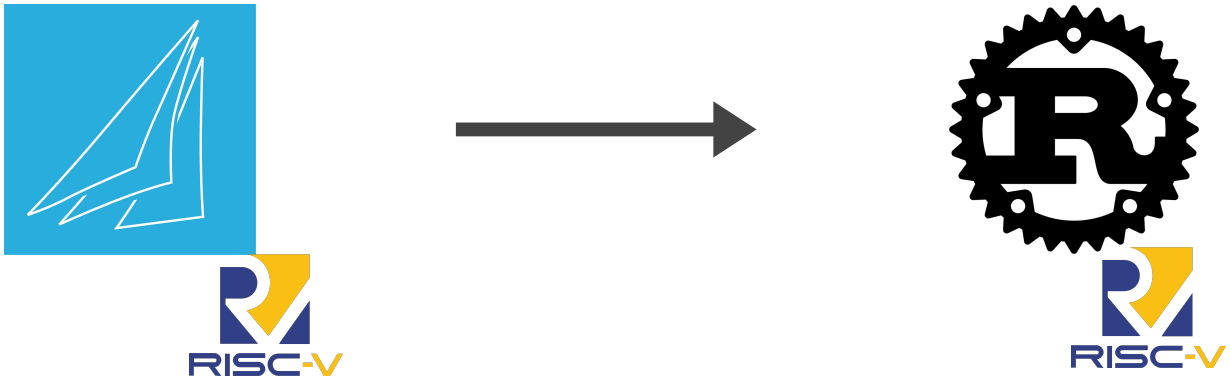
Miralis is written in Rust



The RISC-V spec is written in Sail

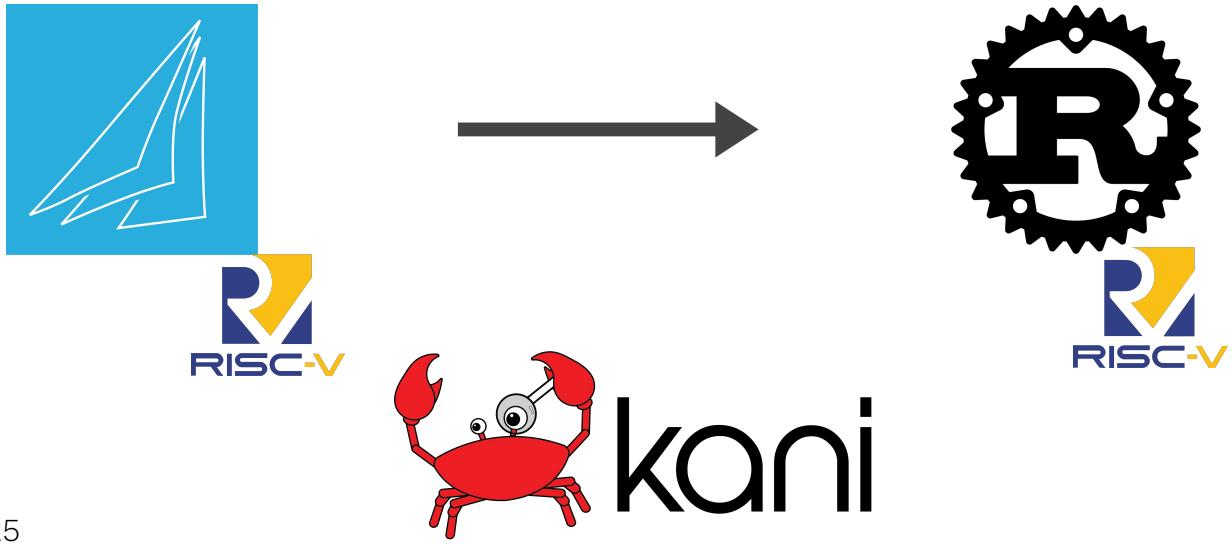
Verifying Miralis - A Rust RISC-V Model

We wrote a Sail-to-Rust backend to generate a Rust RISC-V model



Verifying Miralis - A Rust RISC-V Model

We wrote a Sail-to-Rust backend to generate a Rust RISC-V model

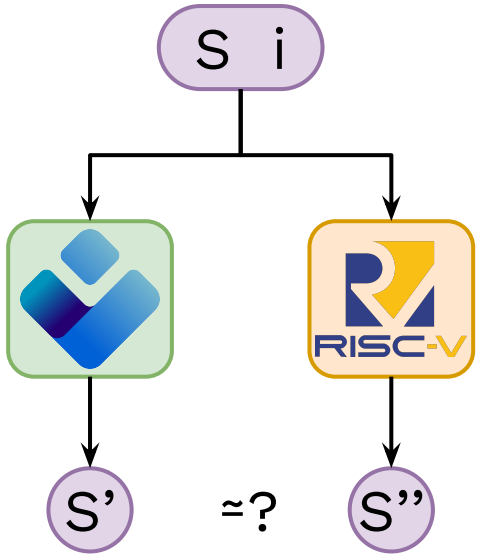


Verifying Miralis - Faithful Emulation of `mret`

```rust
#[cfg_attr(kani, kani::proof)]
pub fn check_mret() {
    let (mut ctx, mut mctx, mut sail_ctx) =
        symbolic::new_symbolic_contexts();

    ctx.emulate_mret(&mut mctx); // Miralis implementation
    execute_MRET(&mut sail_ctx); // Sail implementation

    assert_eq!(
        ctx,
        adapters::sail_to_miralis(sail_ctx, &mctx),
        "mret instruction emulation is not correct"
    );
}
```



Verifying Miralis - Bugs

21 bugs fixed!

Functional Correctness

- mret: set MPP to U-mode after mret
- mvendorid, scounteren, mcountinhibit, mcounteren: 32 bits.
- mepc, sepc: last bits must be 0
- mtvec, stvec: invalid vector modes
- medeleg: bit 11 is read-only zero
- satp: discard invalid writes
- vstart: invalid write mask
- mcause, scause: allow any value
- sie, sip: filter based on mideleg
- Interrupts priority
- and more...

Crash or Sandbox Escape

- pmpaddr: invalid legalization mask
- pmpaddr: W = 1 & R = 0 is reserved
- pmpcfg: out of bound access
- mtvec: PC overflow
- pmpcfg: odd pmpcfg register are invalid

Verifying Miralis - Bugs

Guest access to lock bit

```
0
1     let reg_idx = idx / 8;
2     let inner_idx = idx % 8;
3     let shift = inner_idx * 8; // 8 bits per config
4 -   let cfg = (pmpcfg[reg_idx] >> shift) & 0xff;
5 +   let cfg = (pmpcfg[reg_idx] >> shift) & 0x7f; // Remove the lock
  bit
6     self.set_pmpcfg(idx + offset, cfg as u8);
7
```

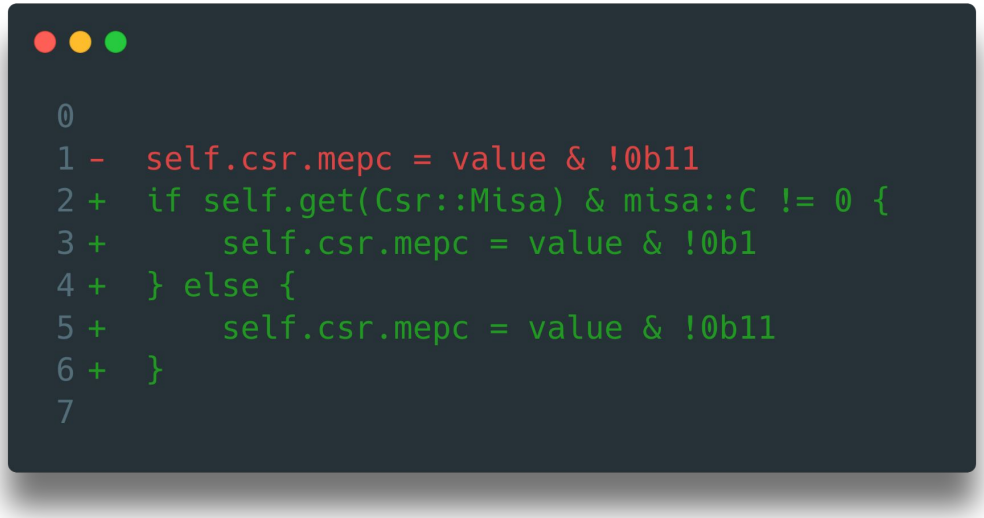
Verifying Miralis - Bugs

Wrong immediate offset in compressed load/stores

```
0
1    C_LW => {
2 -      let imm = (raw >> 3) & 0b100 | (raw >> 7) & 0b111000 | (raw << 1) & 0b1000000;
3 +      let imm = (raw >> 4) & 0b100 | (raw >> 7) & 0b111000 | (raw << 1) & 0b1000000;
4        LoadInstr {
5           rd,
6           rs1,
7           imm,
8        }
9    }
10
```

Verifying Miralis - Bugs

`mepc` mask depends on C extension



```
0
1 -  self.csr.mepc = value & !0b11
2 +  if self.get(Csr::Misa) & misa::C != 0 {
3 +      self.csr.mepc = value & !0b1
4 +  } else {
5 +      self.csr.mepc = value & !0b11
6 +  }
7
```

Lightweight Hypervisor Verification

- There exists official ISA specs
  - Can be leveraged to verify systems properties
- Faithful Emulation + Faithful Execution
  - Reasonable hypervisor spec
- We verified core components of Miralis
  - Instructions emulation, virtual interrupts, memory protection