Provably-Safe Multilingual Software Sandboxing using WebAssembly

Jay Bosamiya, Wen Shih Lim, and Bryan Parno

Carnegie Mellon University

Untrusted Code is Everywhere

Plugins/Extensions

3rd Party Libraries

Modern CDNs

Edge Computing

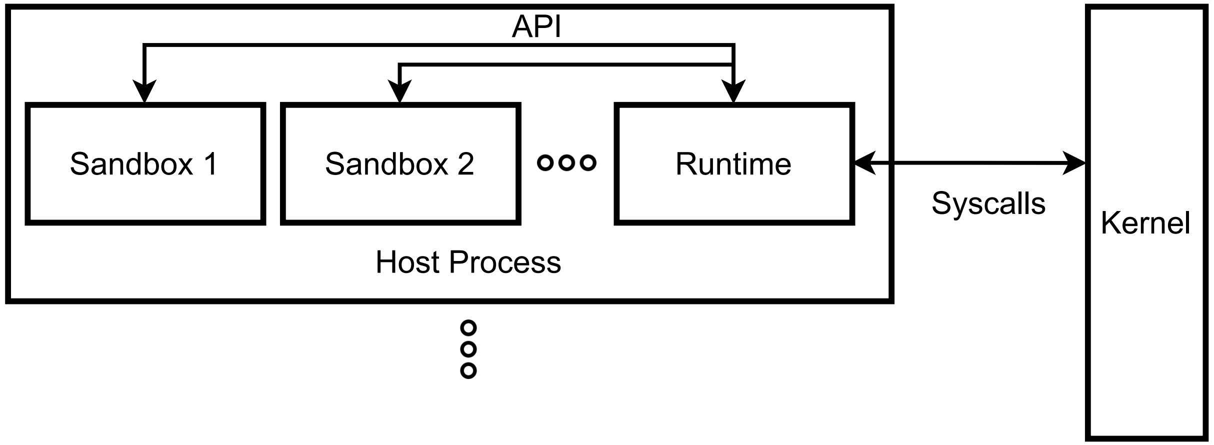
Smart Contracts

The Web

…

Star Wars: Episode II—Attack of the Clones

Intra-Process Sandboxing

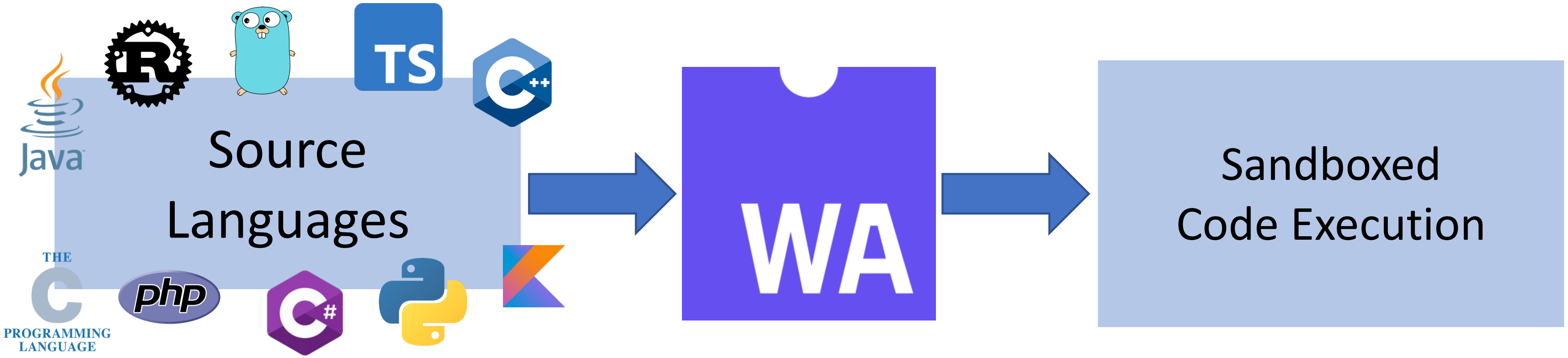


Safety

Performance

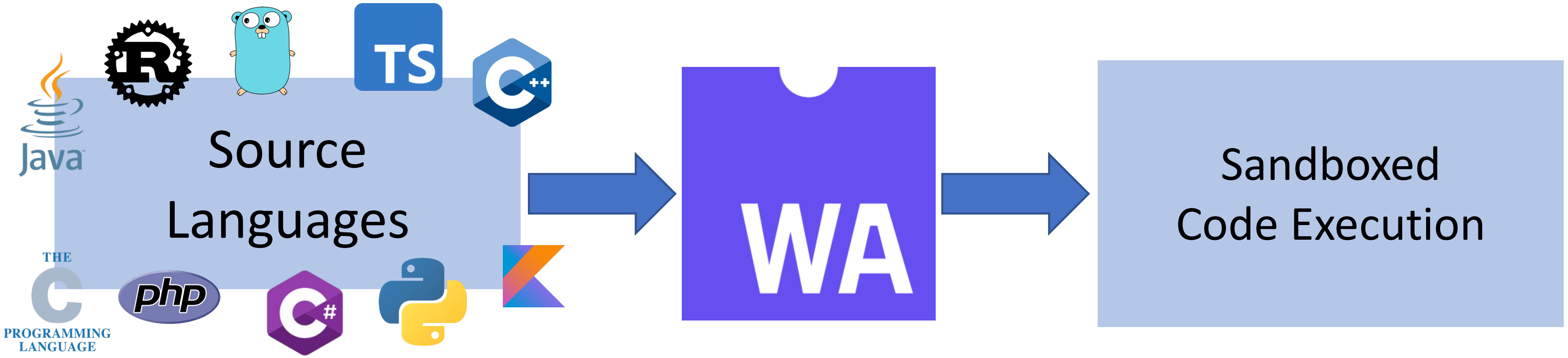
Ease of Use

Sandboxing on the Web



WebAssembly: Promises lightweight, safe & fast execution of untrusted code, on the Web

Sandboxing on the Web, and Beyond



WebAssembly: Promises lightweight, safe & fast execution of untrusted code,
on the Web
(and beyond)

But Promise Only as Strong as Implementation

Our Contributions

Explore two distinct techniques to achieve provably-safe sandboxing

vWasm: formally verified, machine-checked proofs of safety

rWasm: provable safety with competitive performance, *without* writing formal proofs

Brief Tangent: Formal Verification

Mathematical guarantees about software

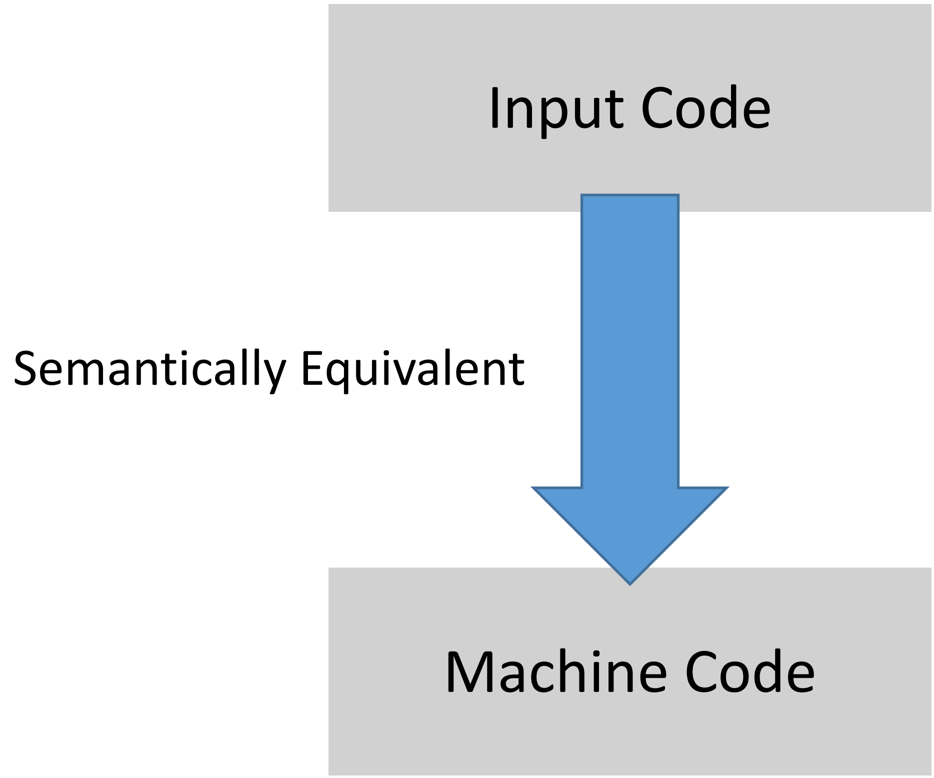
Tools: F*, Dafny, Lean, Coq, …

Specify properties as pre/post conditions, and dependent types

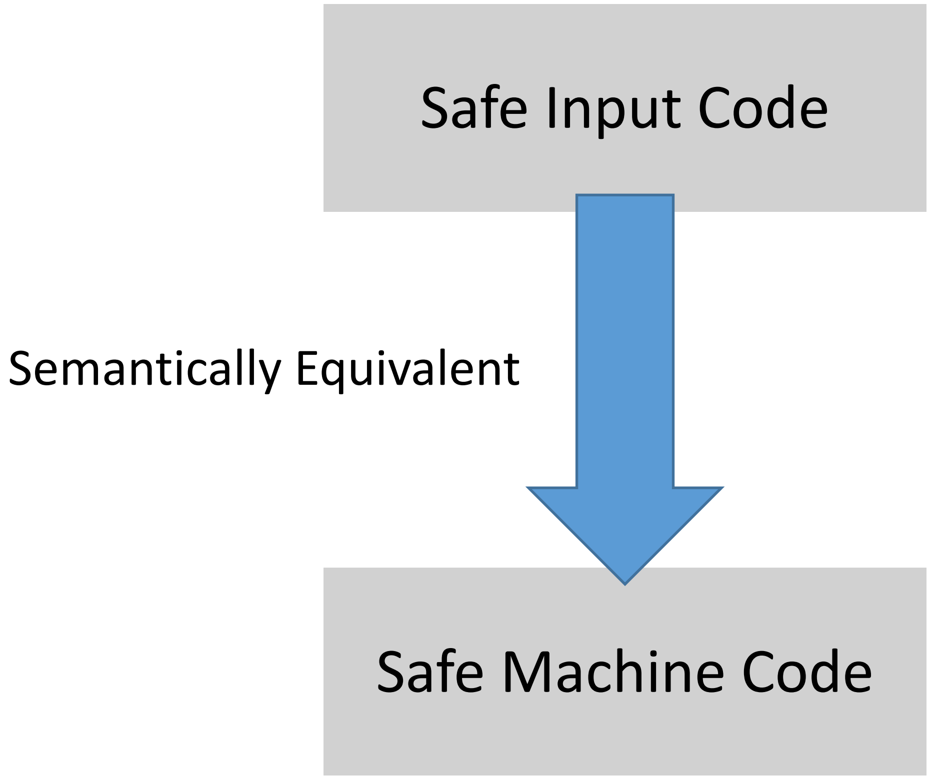
Machine-checked proofs

Assertions checked statically, not at run-time

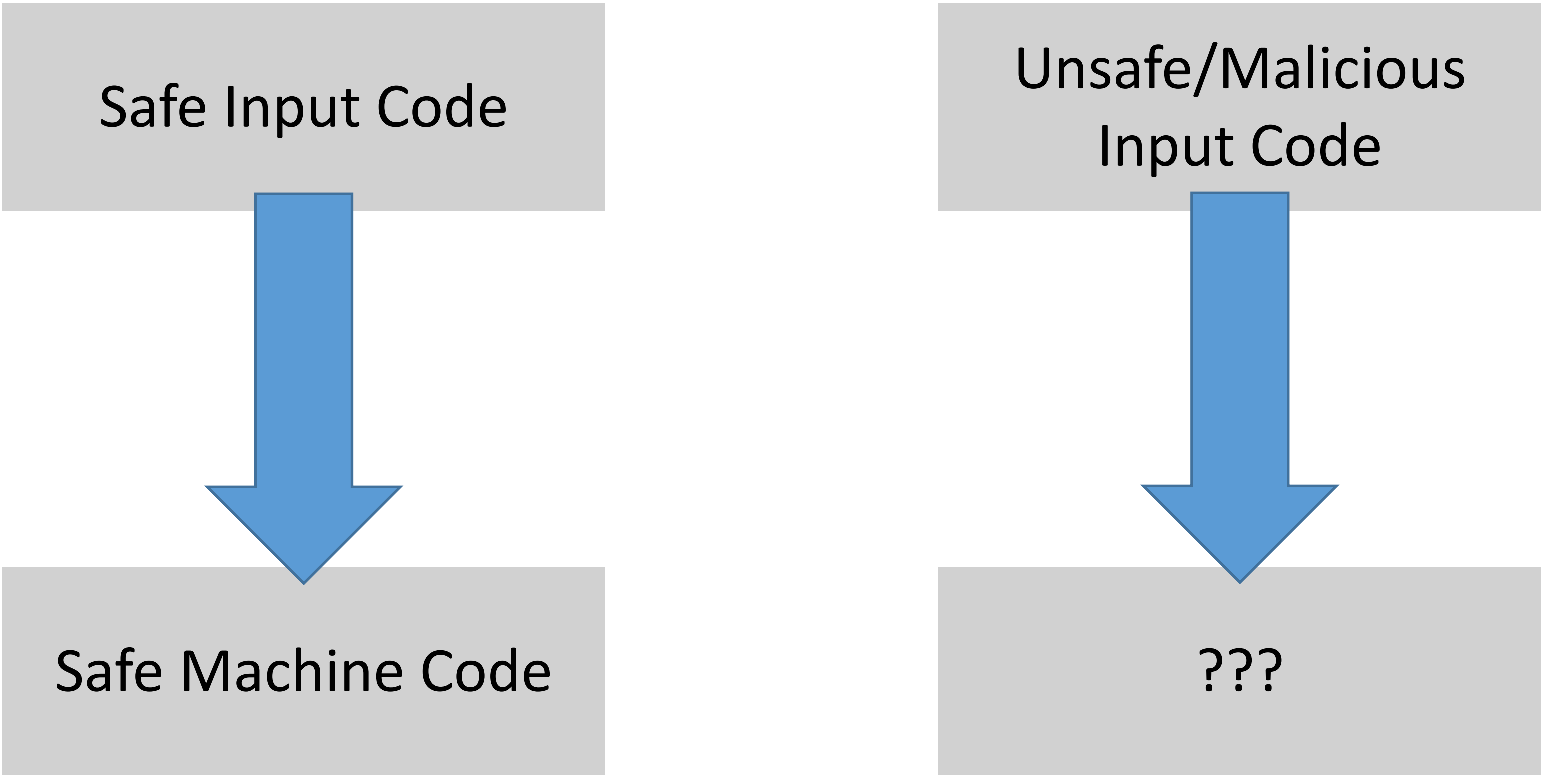
Traditional vs. Sandboxing Verified Compiler



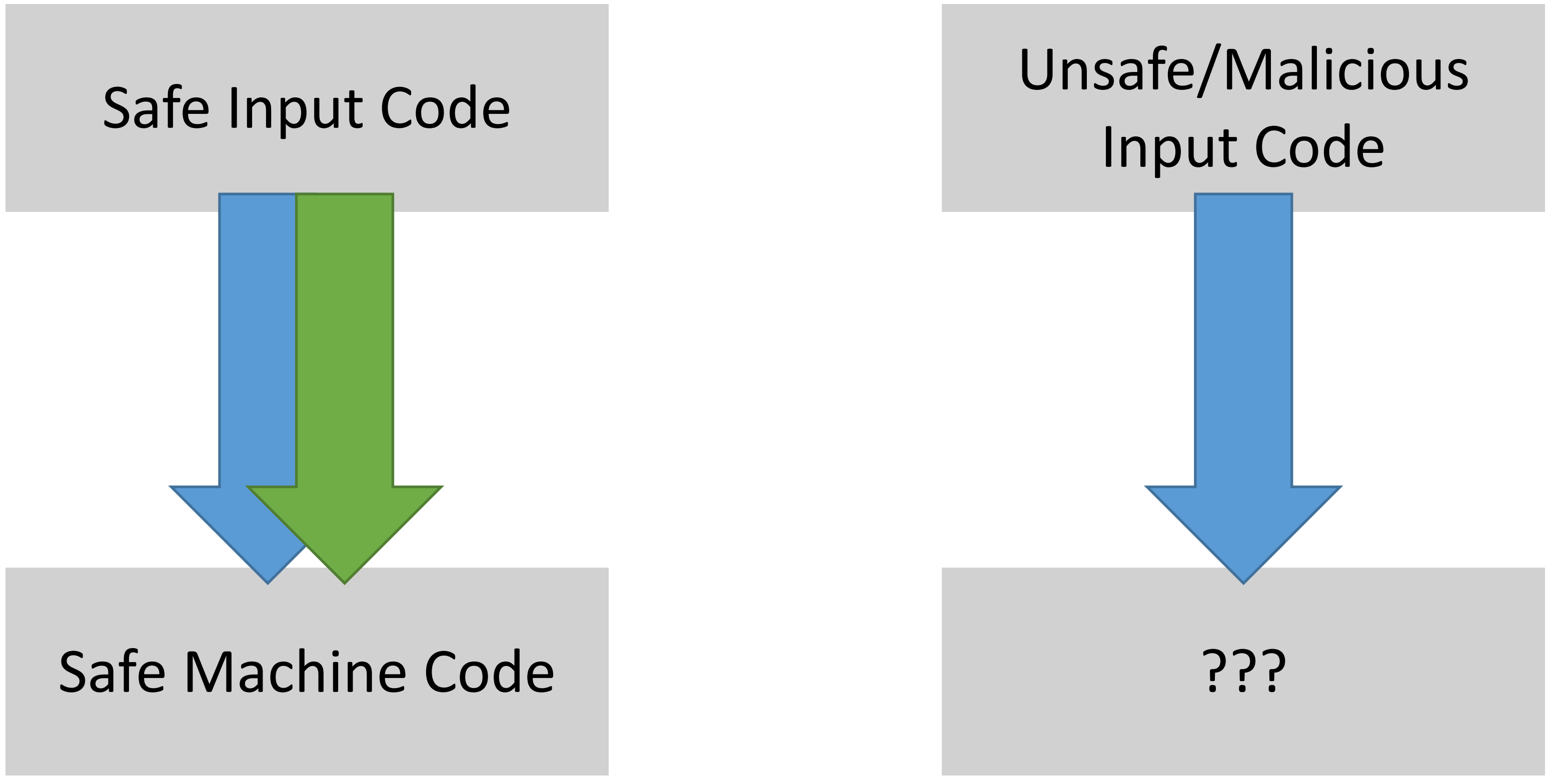
Traditional vs. Sandboxing Verified Compiler



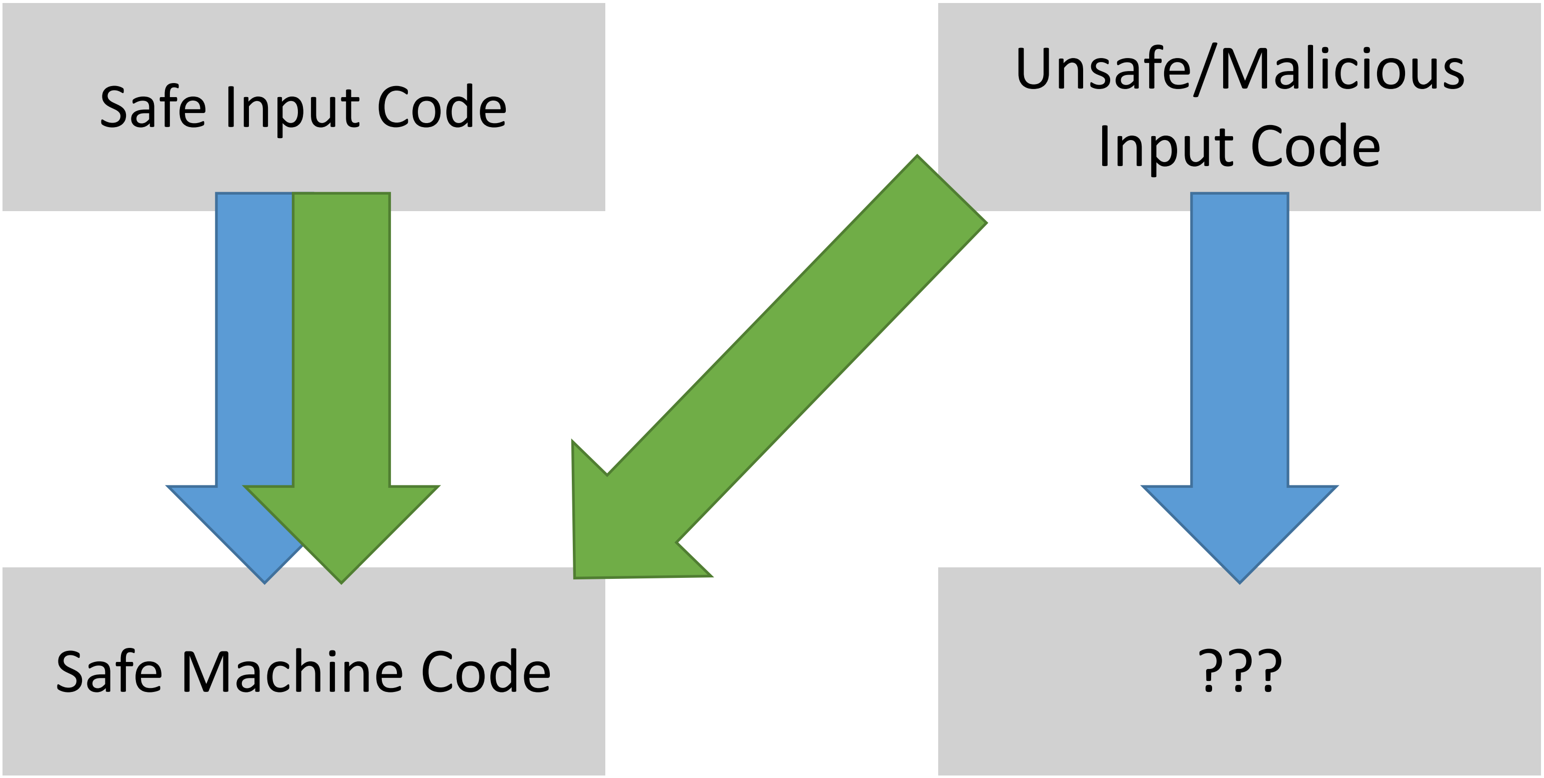
Traditional vs. Sandboxing Verified Compiler



Traditional vs. Sandboxing Verified Compiler



Traditional vs. Sandboxing Verified Compiler



vWasm: Top Level Theorem Statement (simplified)

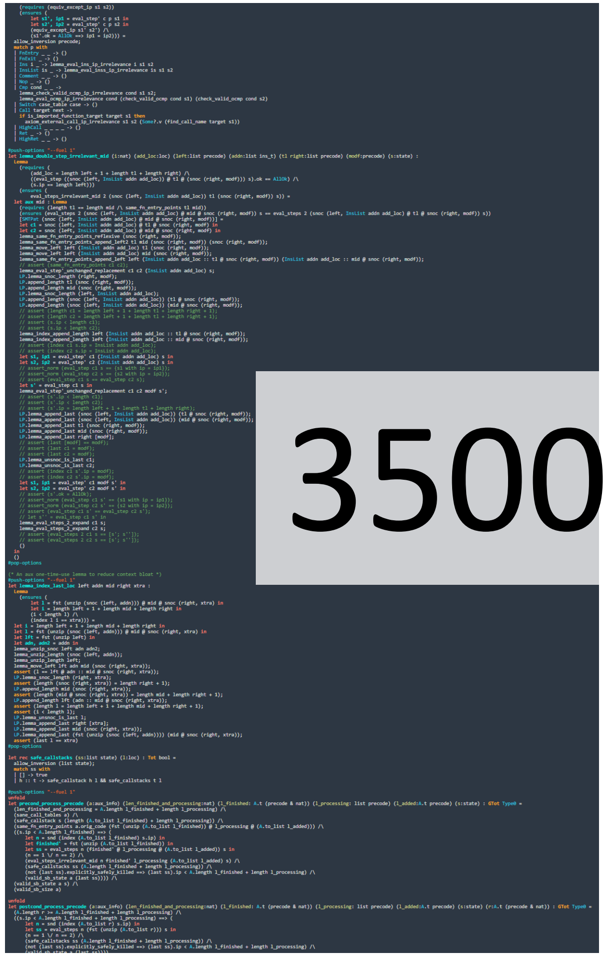
```
val sandbox_compile
  (a:aux) (c:code) (s:erased state) : Err code
    (requires (
      (s.ok = AllOk) ∧
      (reasonable_size a.sb_size s.mem) ∧
      (s.ip `in_code` c) ∧ ...))
    (ensures (λ c' →
      ∀ n. (eval_steps n c' s).ok = AllOk))
```

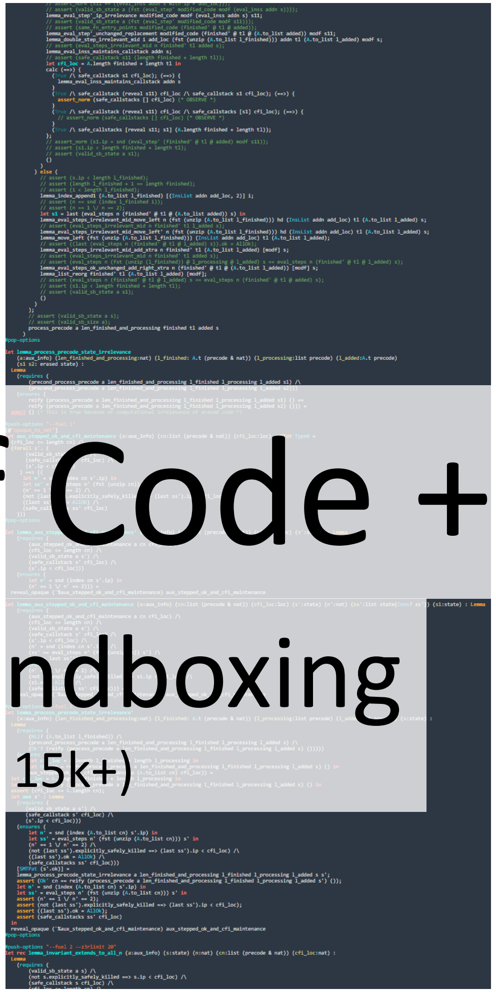
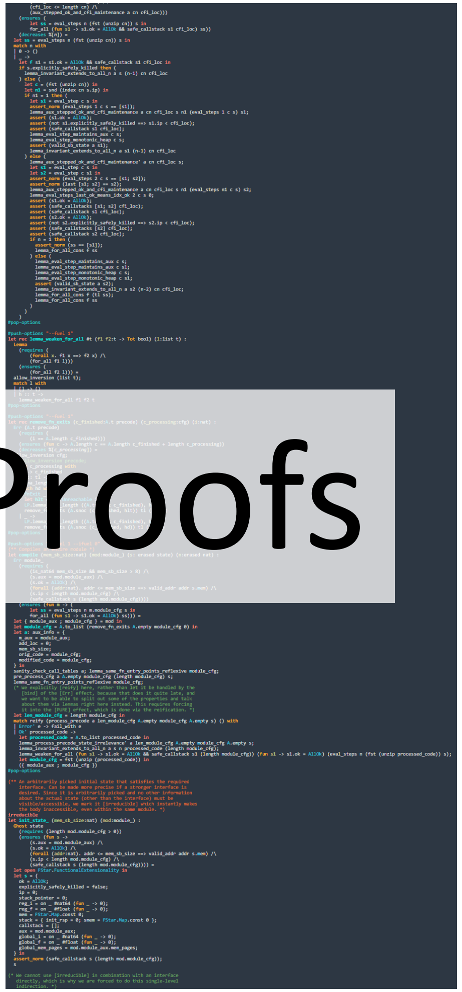
Starting from any "ok" state,
running any number of steps (of the compiled code)
leads to an "ok" state

Only perform explicitly allowed behavior

Prevents:
- OOB memory accesses
- Writing to RO memory
- Calls to unsafe APIs
- …

vWasm: Sandboxing Proof



3500+ Lines of Code + Proofs

For Just Sandboxing

(Overall: 15k+)

vWasm: Sandboxing Proof Sketch

Coarse-Grained Control Flow Integrity

Runtime SFI Checks for Linear Memory, Tables, …

Statically Sized Sandbox

Guarantees w/o Tedium of Formal Proofs

Predictable
Performance

Type/Memory-Safe Language

`#![forbid(unsafe)]`

Lift

Compile

Untrusted Code

Safe Machine Code

rWasm Sandboxing

Memory Safety of Type-Safe Language ⇒ Safe Sandboxing

SFI Checks for Linear Memory, Tables, …

Optimized away at compile-time, whenever possible by `rustc`

Static/Dynamically-Sized Sandbox

rWasm: Runtime Extensions

Inline Reference Monitors

Tracers/Sanitizers

Optimized by `rustc` in tandem with code

rWasm Compilation Example

Compute $\sum_{i=1}^{n} i$

```
(func (param i32) (result i32)
  (local i32)
  loop (result i32)   ;; []
    local.get 0       ;; [a]
    i32.const 1       ;; [a, 1]
    i32.lt_s          ;; [a<1?]
    if (result i32)   ;; []
      local.get 1     ;; [s], return
    else
      local.get 0     ;; [a]
      local.get 1     ;; [a, s]
      i32.add         ;; [a+s]
      local.set 1     ;; [], s ← a + s
      local.get 0     ;; [a]
      i32.const 1     ;; [a, 1]
      i32.sub         ;; [a−1]
      local.set 0     ;; [], a ← a − 1
      br 1            ;; continue
    end
  end)
```

Lift

```
fn func_0(&mut self, a: i32) → Option<i32> {
  let mut local0 = a; let mut local1 = 0i32;
  let mut slot0: TV;  let mut slot1: TV;
  'lbl0: loop {
    slot0 = tv(local0);
    slot1 = tv(1i32);
    slot0 = tv((slot0.vi32()?) <
               (slot1.vi32()?));
    'lbl1: loop {
      if slot0.vi32()? != 0 {
        slot0 = tv(local1);
      } else {
        slot0 = tv(local0);
        slot1 = tv(local1);
        slot0 = tv(slot0.vi32()? +
                   slot1.vi32()?);
        local1 = slot0.vi32()?;
        slot0 = tv(local0);
        slot1 = tv(1i32);
        slot0 = tv(slot0.vi32()? -
                   slot1.vi32()?);
        local0 = slot0.vi32()?;
        continue 'lbl0;
      } break;
    } break;
  }
  Some(slot0.vi32()?)
}
```

Naïve/unoptimized

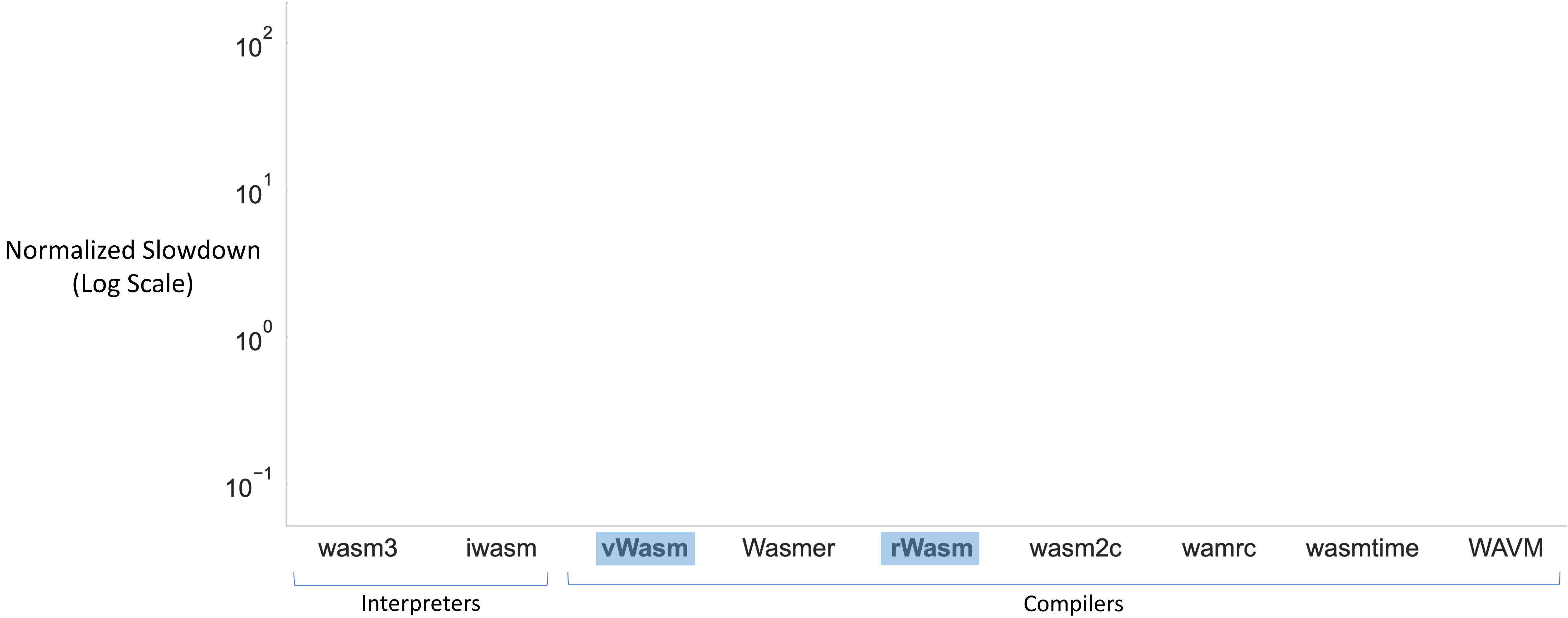
483 Lines of x86-64

Optimized

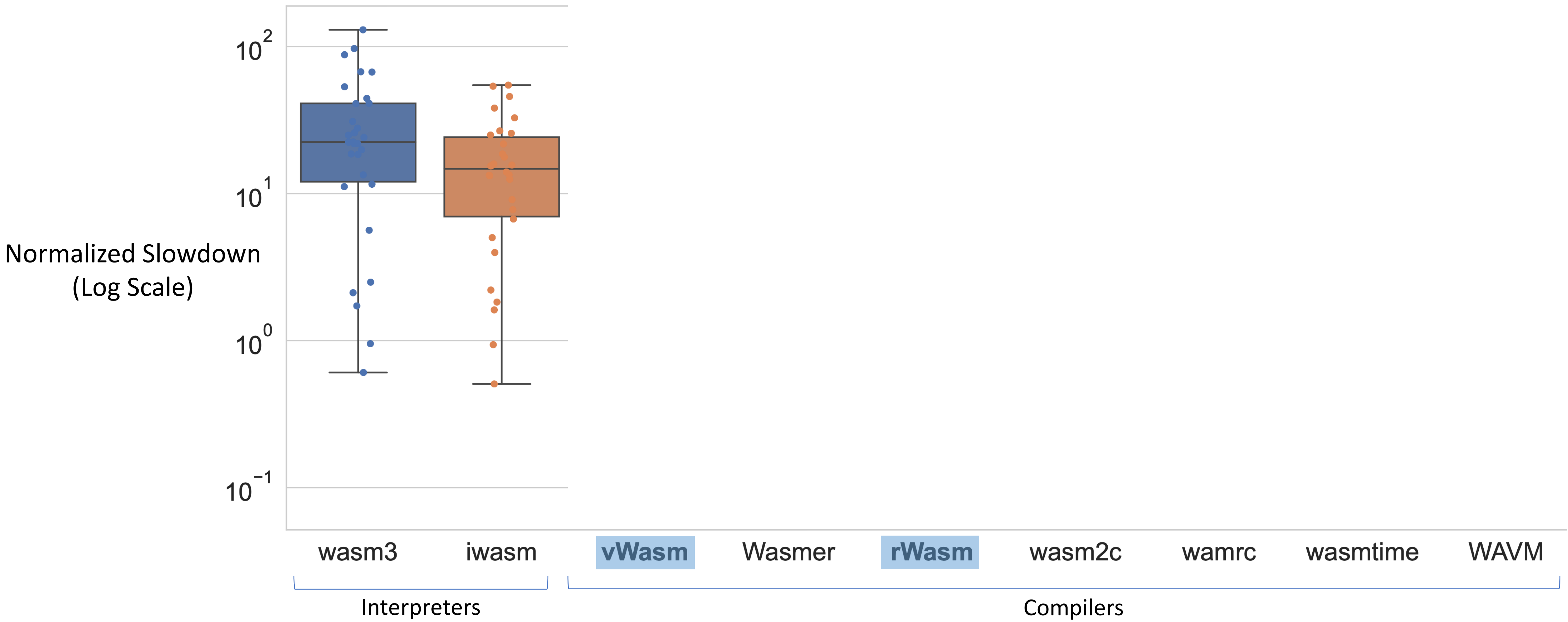
```
func_0:
      test    esi, esi
      jle     .A
      lea     eax, [rsi - 1]
      lea     ecx, [rsi - 2]
      imul    rcx, rax
      mov     edx, eax
      imul    edx, eax
      shr     rcx
      add     edx, esi
      sub     edx, ecx
      mov     eax, 1
      ret
.A:
      xor     edx, edx
      mov     eax, 1
      ret
```

WA → rWasm → Rust → rustc → x86

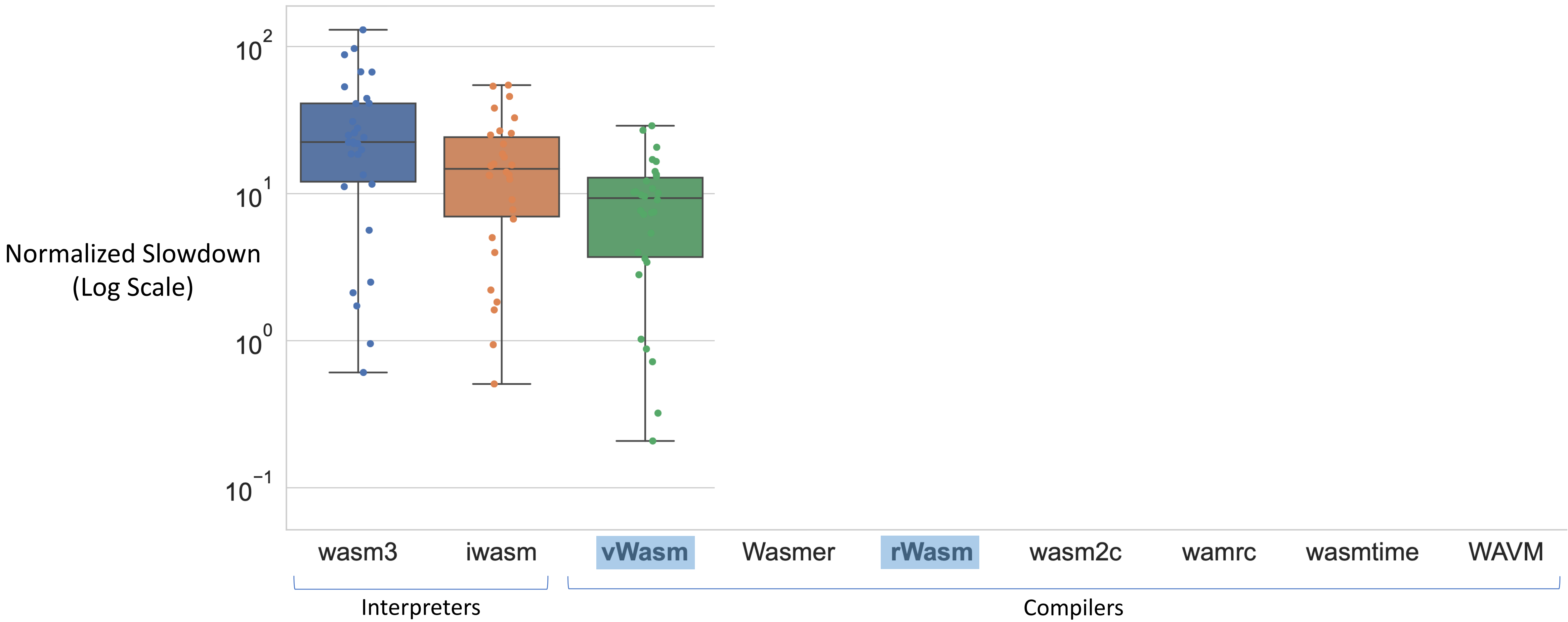
vWasm and rWasm are Competitive



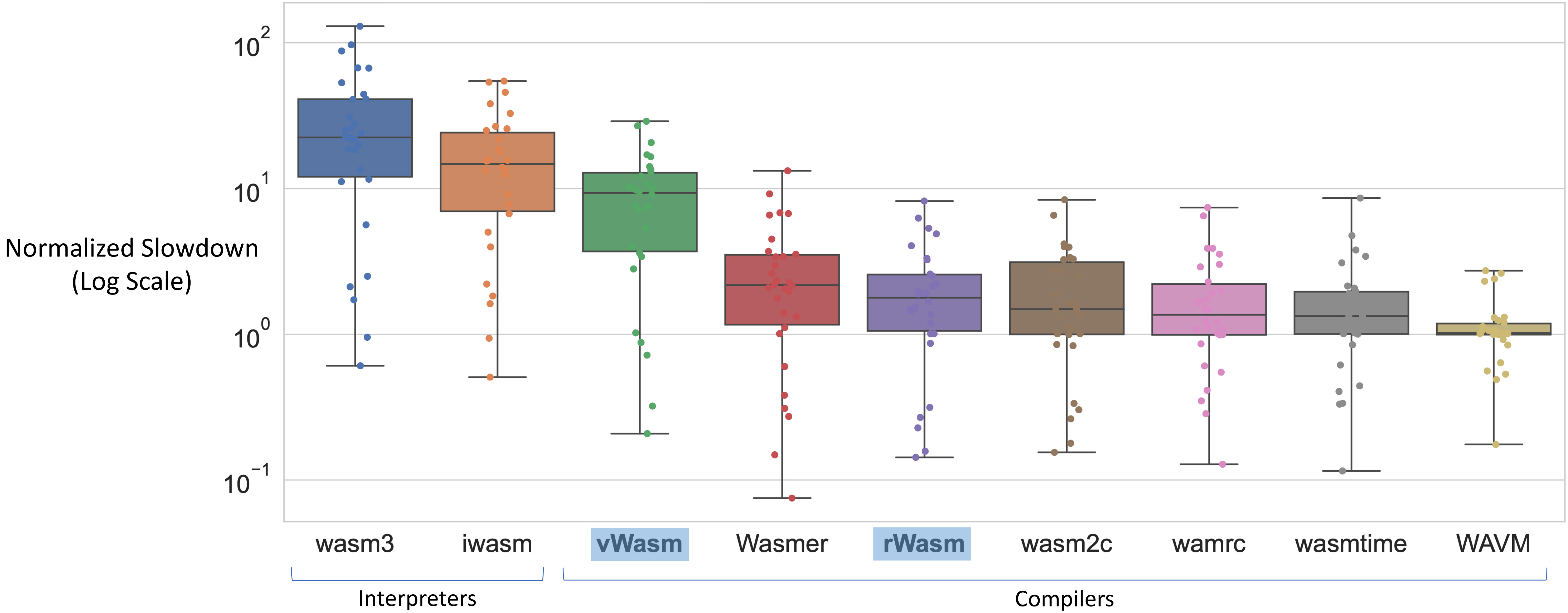
vWasm and rWasm are Competitive



vWasm and rWasm are Competitive



vWasm and rWasm are Competitive



Qualitative Comparison

vWasm	rWasm
Formally Verified w/ Traditional TCB	Portable Across Architectures “For Free”
	Better Execution Speed
Static Property Extensibility	
	Inlined Runtime Extensions
~2 person-years	~1 person-month

Provably-Safe Multilingual Software Sandboxing using WebAssembly

vWasm and rWasm explore two concrete compelling points in design space, with various tradeoffs

High-performance and strong safety are not mutually exclusive goals

Interesting space for further exploration

https://github.com/secure-foundations/{rWasm,vWasm,wasm-semantics-fuzzer,provably-safe-sandboxing-wasm-usenix22}

jaybosamiya@cmu.edu / https://www.jaybosamiya.com/