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
...

Software Sandboxing  WebAssembly  vWasm  rWasm  Evaluation  Tradeoffs
Intra-Process Sandboxing

Safety

Performance  Ease of Use

Software Sandboxing  WebAssembly  vWasm  rWasm  Evaluation  Tradeoffs
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

Input Code

Semantically Equivalent

Machine Code
Traditional vs. Sandboxing Verified Compiler

Safe Input Code

Semantically Equivalent

Safe Machine Code
Traditional vs. Sandboxing Verified Compiler

Safe Input Code

Unsafe/Malicious Input Code

Safe Machine Code

???
Traditional **vs. Sandboxing** Verified Compiler

![Diagram showing the comparison between safe and unsafe/malicious input codes, leading to safe machine code and a question mark for unsafe/malicious input code.]

- **Safe Input Code**: Safe Machine Code
- **Unsafe/Malicious Input Code**: ???
Traditional vs. Sandboxing Verified Compiler

Safe Input Code

Safe Machine Code

Unsafe/Malicious Input Code

???
vWasm: Top Level Theorem Statement (simplified)

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

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

Only perform explicitly allowed behavior

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

Software Sandboxing
WebAssembly
vWasm
rWasm
Evaluation
Tradeoffs
vWasm: Sandboxing Proof

3500+ Lines of Code + Proofs
For Just Sandboxing
(Overall: 15k+)

Software Sandboxing  WebAssembly  vWasm  rWasm  Evaluation  Tradeoffs
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

Type/Memory-Safe Language
#![forbid(unsafe)]

Untrusted Code  Safe Machine Code

Predictable Performance

Lift  Compile
rWasm Sandboxing

Memory Safety of Type-Safe Language $\Rightarrow$ 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 $\sum_{i=1}^{n} i$

```rust
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;
    \l10: loop {
        slot0 = tv(local0);
        slot1 = tv(i32); \l11: loop {
            if slot0.vi32() ?? 10 < slot1.vi32() ?? 10 {
                slot0 = tv(local0);
                slot1 = tv(i32);
            } else {
                slot0 = tv(local0);
                slot1 = tv(i32);
            }
        }
    }
}
```

483 Lines of x86-64

Naive/unoptimized

Optimized

Software Sandboxing  WebAssembly  vWasm  rWasm  Evaluation  Tradeoffs
vWasm and rWasm are Competitive
vWasm and rWasm are Competitive
vWasm and rWasm are Competitive

Normalized Slowdown (Log Scale)

Interpreters

Compilers

wasm3 iwasm vWasm Wasmer rWasm wasm2c wamrc wasmtime WAVM

Software Sandboxing WebAssembly vWasm rWasm Evaluation Tradeoffs
vWasm and rWasm are Competitive
Qualitative Comparison

**vWasm**
- Formally Verified w/ Traditional TCB
- Static Property Extensibility
- ~2 person-years

**rWasm**
- Portable Across Architectures “For Free”
- Better Execution Speed
- Inlined Runtime Extensions
- ~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/