Building a safety verifier for Wasm

Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, Deian Stefan

WebAssembly

- Platform independent bytecode used in and out of the browser
  - Supported by all major browsers
  - Can be targeted by most major languages

- Can be compiled to native code to improve performance
  - Fastly CDN AOT compiles Wasm modules for deployment
  - Firefox AOT-compiles 2 of its media processing libraries from Wasm
  - Microsoft Flight Simulator deploys some of its code as AOT-compiled Wasm

WebAssembly security

- WebAssembly modules are isolated — they never access outside their assigned address space.

- Wasm-to-native compiles guarantee isolation by inserting dynamic safety checks into generated native code
  - Memory accesses are checked to be in bounds
  - Indirect jumps and calls are checked to point to valid code

- Safety checks are inserted *before* optimization

Compilation gone wrong

```
...
for(int i = 0; i < 10; i++){
    switch(casenum){ … }
}
```

Not Optimized

```
xor rcx rcx;
Loop_Start: cmp rax, 0x7;
jae default_case;
mov rdx, jump_table_base;
mov rbx, [rdx + rax * 4];
add rdx, rbx; jump to the target
jmp rdx;
…
add ecx, 1
cmp ecx, 10
jle Loop_Start
```

Compilation gone wrong

```
...
for(int i = 0; i < 10; i++){
    switch(casenum){ … }
}
```

Not Optimized

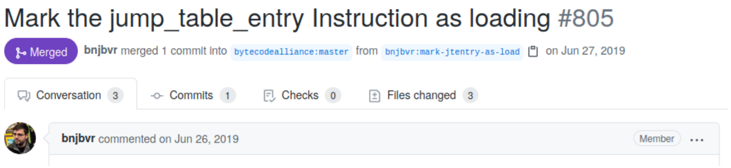
```
xor rcx rcx;
Loop_Start: cmp rax, 0x7;
jae default_case;
mov rdx, jump_table_base;
mov rbx, [rdx + rax * 4];
add rdx, rbx; jump to the target
jmp rdx;
…
add ecx, 1
cmp ecx, 10
jle Loop_Start
```

Optimized

```
xor rcx rcx;
mov rdx, jump_table_base;
mov rbx, [rdx + rax * 4];
Loop_Start: cmp rax, 0x7;
jae default_case;
add rdx, rbx; jump to the target
jmp rdx;
…
add ecx, 1
cmp ecx, 10
jle Loop_Start
```

Compilation gone wrong

```
...
for(int i = 0; i < 10; i++){
    switch(casenum){ … }
}
```



Not Optimized

```
xor rcx rcx;
Loop_Start: cmp rax, 0x7;
jae default_case;
mov rdx, jump_table_base;
mov rbx, [rdx + rax * 4];
add rdx, rbx; jump to the target
jmp rdx;
…
add ecx, 1
cmp ecx, 10
jle Loop_Start
```

Optimized

```
xor rcx rcx;
mov rdx, jump_table_base;
mov rbx, [rdx + rax * 4];
Loop_Start: cmp rax, 0x7;
jae default_case;
add rdx, rbx; jump to the target
jmp rdx;
…
add ecx, 1
cmp ecx, 10
jle Loop_Start
```

What went wrong?

- Safety checks are inserted before compiler optimizations run for performance reasons.

- Compiler passes can move or wrongly ellide these checks in such a way that unsafe behavior is allowed.

- This can break isolation, and potentially allow unsafe code to run.

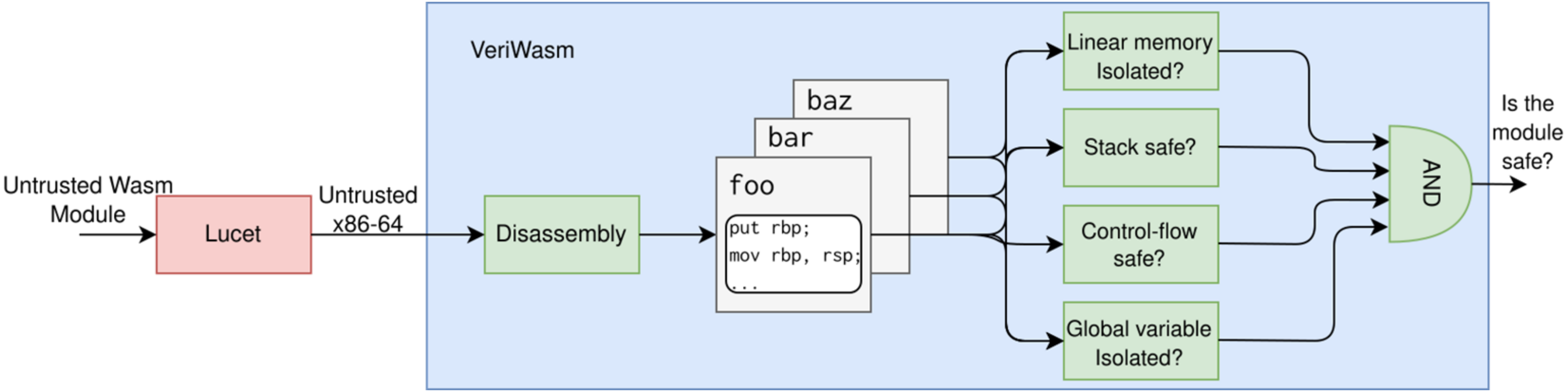
Goal: Check whether AOT-compiled Wasm is safe

- Building a verified compiler is labor-intensive
    - Compcert required over 100,000 lines of code and 6 person years to complete

- Instead: check whether Wasm code is safe, post-compilation

VeriWasm

- Checks untrusted x86 module output by compiler
- Safety properties checked for each function
- Outputs isolation judgement for full binary



Verifying the safety of natively-compiled Wasm

- What does VeriWasm check?
- How does VeriWasm check it?
- How do we know VeriWasm is correct?

Verifying the safety of natively-compiled Wasm

- **What does VeriWasm check?**
- How does VeriWasm check it?
- How do we know VeriWasm is correct?

What does VeriWasm check?

- Isolation: For all possible executions of the module, the module never accesses memory outside it’s address space or otherwise executes unsafe code.

What does VeriWasm check?

- Isolation: For all possible executions of the module, the module never accesses memory outside it’s address space or otherwise executes unsafe code.

- Problem: verifying isolation of arbitrary binaries is at worst undecidable, and at best complex and not scalable

What does VeriWasm check?

- Isolation: For all possible executions of the module, the module never accesses memory outside it's address space or otherwise executes unsafe code.

- Problem: verifying isolation of arbitrary binaries is at worst undecidable, and at best complex and not scalable

- Two key insights that simplify analysis:
  - We can take advantage of language-level restrictions of Wasm
  - We can break down the isolation property into simpler safety subproperties that together prove isolation

Insight 1: Take advantage of Wasm structure

- Code generated from Wasm only represents a subset of x86-64
- Some code constructs like arbitrary computed jumps are not representable in Wasm

Insight 1: Take advantage of Wasm structure

- Code generated from Wasm only represents a subset of x86-64
- Some code constructs like arbitrary computed jumps are not representable in Wasm

```
local.get localidx
local.set localidx
```

WebAssembly

`rsp = rsp ± c`	stack adjustments
`x = mem[rsp ± c]`	stack loads
`mem[rsp ± c] = x`	stack stores

X86-64

Insight 2: Break isolation into simpler properties

- Isolation: For all possible executions of the module, the module never accesses memory outside it's address space or otherwise executes unsafe code.

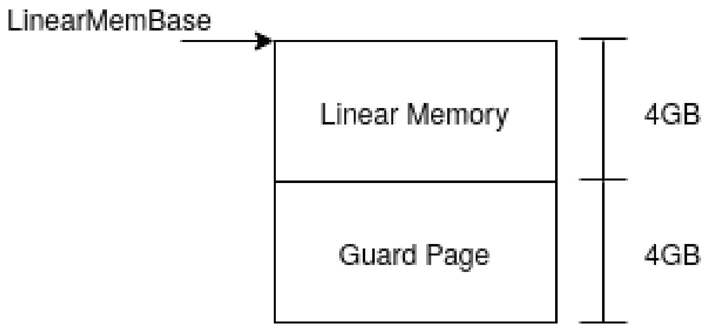
Insight 2: Break isolation into simpler properties

- Isolation: For all possible executions of the module, the module never accesses memory outside it’s address space or otherwise executes unsafe code.
- Instead: prove simpler properties that together prove isolation

Feature	**Safety property**	**Description**
Linear memory	Linear memory isolation	All linear memory reads and writes fall within the 4GB linear memory space (or surrounding guard pages).
Stack	Stack isolation	Stack reads fall within the stack region (or surrounding guard pages).
	Stack-frame integrity	Stack writes are to local variables in the current stack frame.
Global variables	Global variable isolation	Global variable accesses fall within the global variable memory region.
Control flow	Jump target validity	All indirect jumps target valid code blocks.
	Call target validity	All indirect calls target valid functions.
	Return target validity	Functions return to their respective call sites.

Example safety property: linear memory safety

- Invariant 1: All linear memory accesses fall in LinearMemBase + 8GB region
    - Show that all accesses are of the form:
      mem[LinearMemBase + x + y] where x <= 2^32 and y <= 2^32
- Invariant 2: At every function call, the RDI register is LinearMemBase

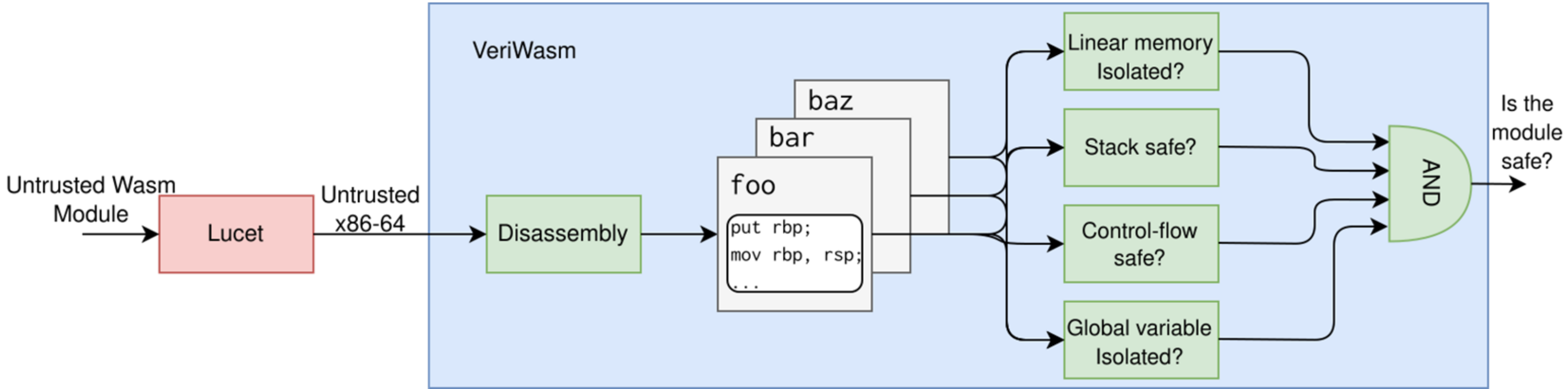


Verifying the safety of natively-compiled Wasm

- What does VeriWasm check?
- **How does VeriWasm check it?**
- How do we know VeriWasm is correct?

Analysis passes

- Each function is analyzed independently
  - Simplifies analysis
  - Allows for checking in parallel
- Analysis based on abstract interpretation
- Track state of variables in registers and on the stack



Heap analysis example

```
foo:
; ASSUME: rdi is LinearMemBase
; TRACK: rax, rbx, ... are Unknown
  ...
  mov eax, eax;
  ; TRACK: rax Bounded
  mov rsi, [rdi + rax + 0x48];
  ; ASSERT: rdi is LinearMemBase
  ; ASSERT: rax and 0x48 are Bounded
  ...
  call bar;
  ; ASSERT: rdi is LinearMemBase
  ...
```

Heap analysis example

```
foo:
; ASSUME: rdi is LinearMemBase
; TRACK: rax, rbx, ... are Unknown
  ...
  mov eax, eax;
  ; TRACK: rax Bounded
  mov rsi, [rdi + rax + 0x48];
  ; ASSERT: rdi is LinearMemBase
  ; ASSERT: rax and 0x48 are Bounded
  ...
  call bar;
  ; ASSERT: rdi is LinearMemBase
  ...
```

Heap analysis example

```
foo:
; ASSUME: rdi is LinearMemBase
; TRACK: rax, rbx, ... are Unknown
  ...
  mov eax, eax;
  ; TRACK: rax Bounded
  mov rsi, [rdi + rax + 0x48];
  ; ASSERT: rdi is LinearMemBase
  ; ASSERT: rax and 0x48 are Bounded
  ...
  call bar;
  ; ASSERT: rdi is LinearMemBase
  ...
```

Heap analysis example

```
foo:
; ASSUME: rdi is LinearMemBase
; TRACK: rax, rbx, ... are Unknown
   ...
   mov eax, eax;
   ; TRACK: rax Bounded
   mov rsi, [rdi + rax + 0x48];
   ; ASSERT: rdi is LinearMemBase
   ; ASSERT: rax and 0x48 are Bounded
   ...
   call bar;
   ; ASSERT: rdi is LinearMemBase
   ...
```

Heap analysis example

```
foo:
; ASSUME: rdi is LinearMemBase
; TRACK: rax, rbx, ... are Unknown
  ...
  mov eax, eax;
  ; TRACK: rax Bounded
  mov rsi, [rdi + rax + 0x48];
  ; ASSERT: rdi is LinearMemBase
  ; ASSERT: rax and 0x48 are Bounded
  ...
  call bar;
  ; ASSERT: rdi is LinearMemBase
  ...
```

Verifying the safety of natively-compiled Wasm

- What does VeriWasm check?
- How does VeriWasm check it?
- **How do we know VeriWasm is correct?**

Verification

- We verify in the Coq theorem prover:
    - That proving all our subproperties implies isolation
    - That our verification algorithm is sound

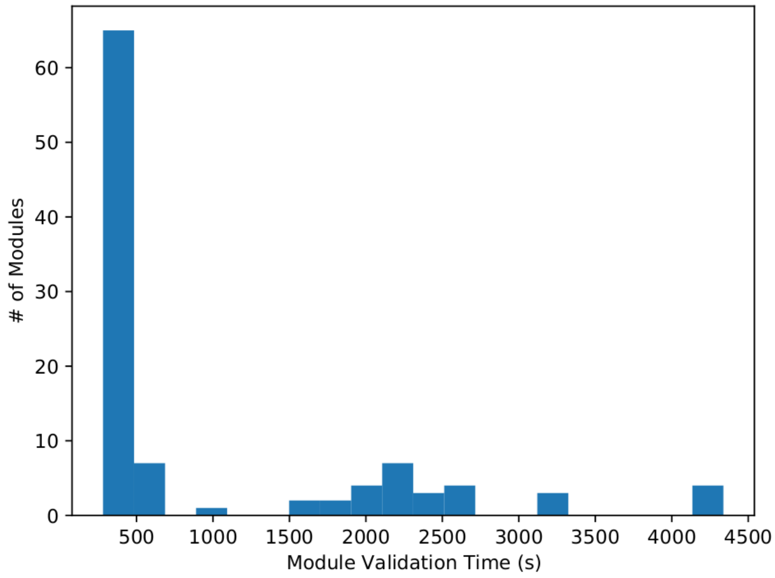
- Verification uncovered several bugs in our implementation:
    - RDI (the register designated to hold the heap base) needs to point to the base of the heap at each call
    - VeriWasm must compensate for the fact that function calls may not save callee-saved registers

Evaluating VeriWasm

- We verified several libraries:
  - 2 firefox libraries currently shipped as natively-compiled Wasm
  - Spec2006 benchmarks (or subset that we can compile to Wasm)
  - Lucet’s microbenchmark suite
- Verified 101 executables on Fastly’s edge computing platform
- Rediscovered bugs in other SFI systems

Evaluation performance

- Validates ~10 functions a second
- Firefox libraries require less than 3 minutes to validate each
- Fastly binaries require median of 6 minutes 30 seconds



Summary

- VeriWasm can verify that Wasm modules compiled to native code are safe.
- It does this by splitting isolation into simpler properties and verifying these simpler properties
- We verify our verification algorithm using the Coq theorem prover