RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes

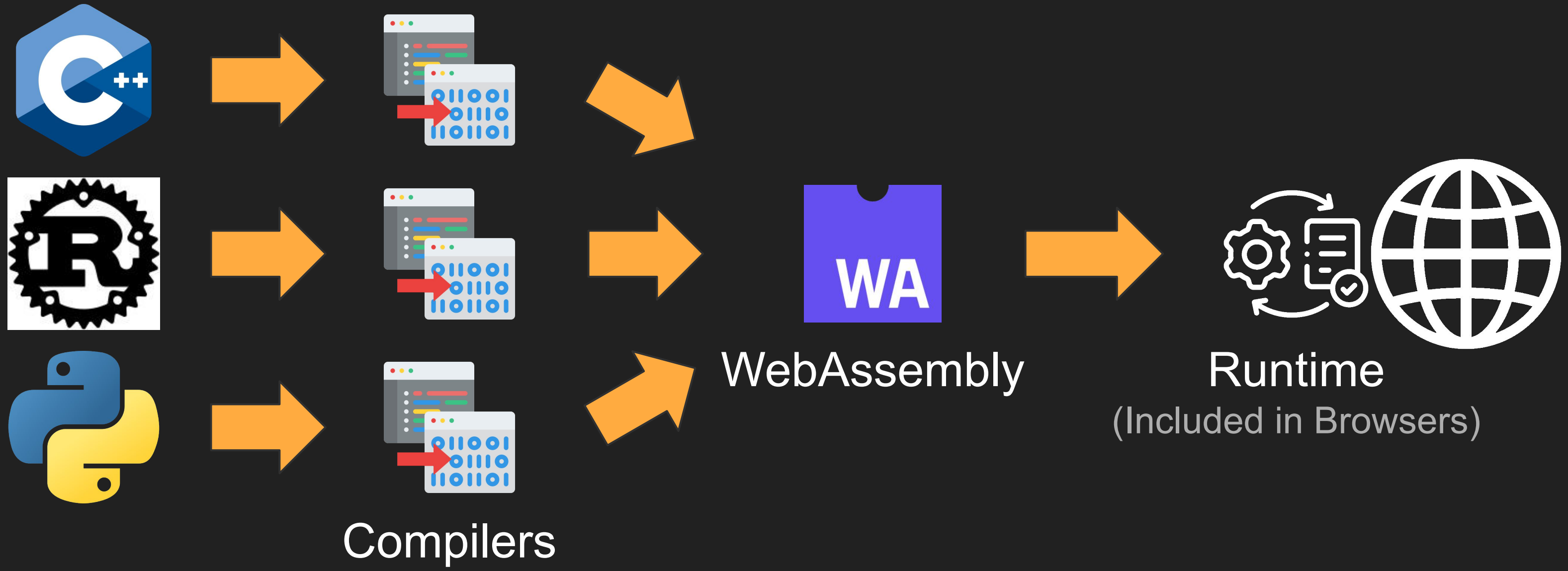
Junyoung Park[1], Yunho Kim*[2], Insu Yun*[1]

[1]KAIST, [2]Hanyang University
*Co-corresponding Authors

WebAssembly (WASM)

- **Fast, safe, portable, and compact** language

- Best for compilation target for other languages



WebAssembly Runtimes

- WebAssembly runs on a stack machine
- Stack machine is slow → Let's compile the code!
- Just-In-Time (JIT) compilation to machine code

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

→ Compile

```
push rbp
mov rbp, rsp
not rcx
mov rax, rdx
or rax, rcx
mov rsp, rbp
pop rbp
ret
```

Compiler Optimizations

- Optimizations to further boost speed

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

1. Translate to IR

```
[Args] v0: i64, v1: i64

v2 = band v0, v1
v3 = iconst.i64 -1
v4 = bxor v1, v3
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Apply simple rule: v1 ^ -1 → ~v1 (changed to not)

```
[Args] v0: i64, v1: i64

v2 = band v0, v1
v3 = iconst.i64 -1
v4 = bxor v1, v3
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Apply simple rule: v1 ^ -1 → ~v1 (changed to not)

```
[Args] v0: i64, v1: i64

v2 = band v0, v1
v3 = iconst.i64 -1
v4 = bxor v1, v3
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Apply simple rule: v1 ^ -1 → ~v1 (changed to not)

2. Simple rule
v1 ^ -1 → ~v1

```
[Args] v0: i64, v1: i64

v2 = band v0, v1
v3 = iconst.i64 -1
v4 = bxor v1, v3
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Apply simple rule: v1 ^ -1 → ~v1 (changed to not)

```
[Args] v0: i64, v1: i64

v2 = band v0, v1

v4 = bnot v1
v5 = bor v2, v4
return v5
```

2. Simple rule
v1 ^ -1 → ~v1

```
[Args] v0: i64, v1: i64

v2 = band v0, v1
v3 = iconst.i64 -1
v4 = bxor v1, v3
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Can also apply complex rule: (v0 & v1) | ~v1 → v0 | ~v1

```
[Args] v0: i64, v1: i64

v2 = band v0, v1

v4 = bnot v1
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Can also apply complex rule: (v0 & v1) | ~v1 → v0 | ~v1

```
[Args] v0: i64, v1: i64

v2 = band v0, v1

v4 = bnot v1
v5 = bor v2, v4
return v5
```

Compiler Optimizations

- Can also apply complex rule: (v0 & v1) | ~v1 → v0 | ~v1

```
[Args] v0: i64, v1: i64

v2 = band v0, v1

v4 = bnot v1
v5 = bor v2, v4
return v5
```

3. Apply complex rule
(v0 & v1) | ~v1
→ v0 | ~v1

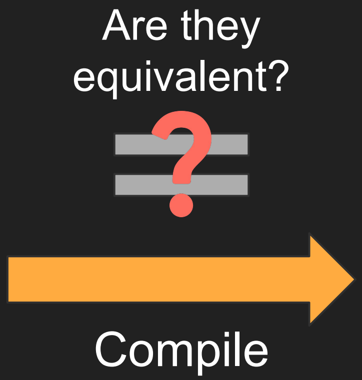
```
[Args] v0: i64, v1: i64

v2 = v0

v4 = bnot v1
v5 = bor v2, v4
return v5
```

Semantic Bugs

- What happens if optimization rules are wrongly written?

- Semantic bug: For some input, exec. of original code != exec. of compiled code

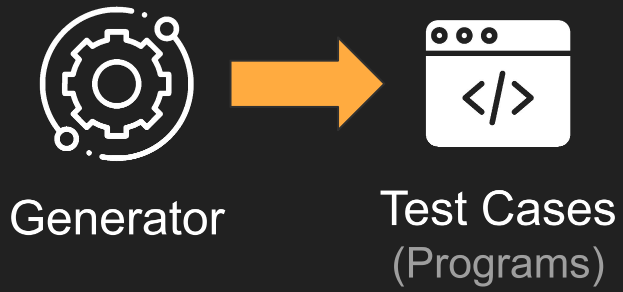
```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```




```
push rbp
mov rbp, rsp
not rcx
mov rax, rdx
or rax, rcx
mov rsp, rbp
pop rbp
ret
```

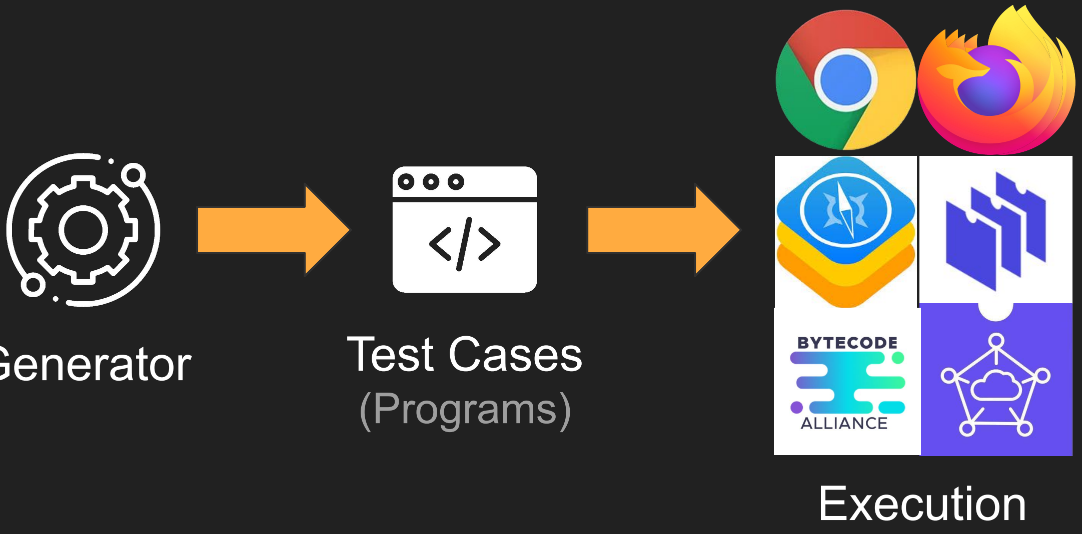
Finding Semantic Bugs

- Differential fuzzing



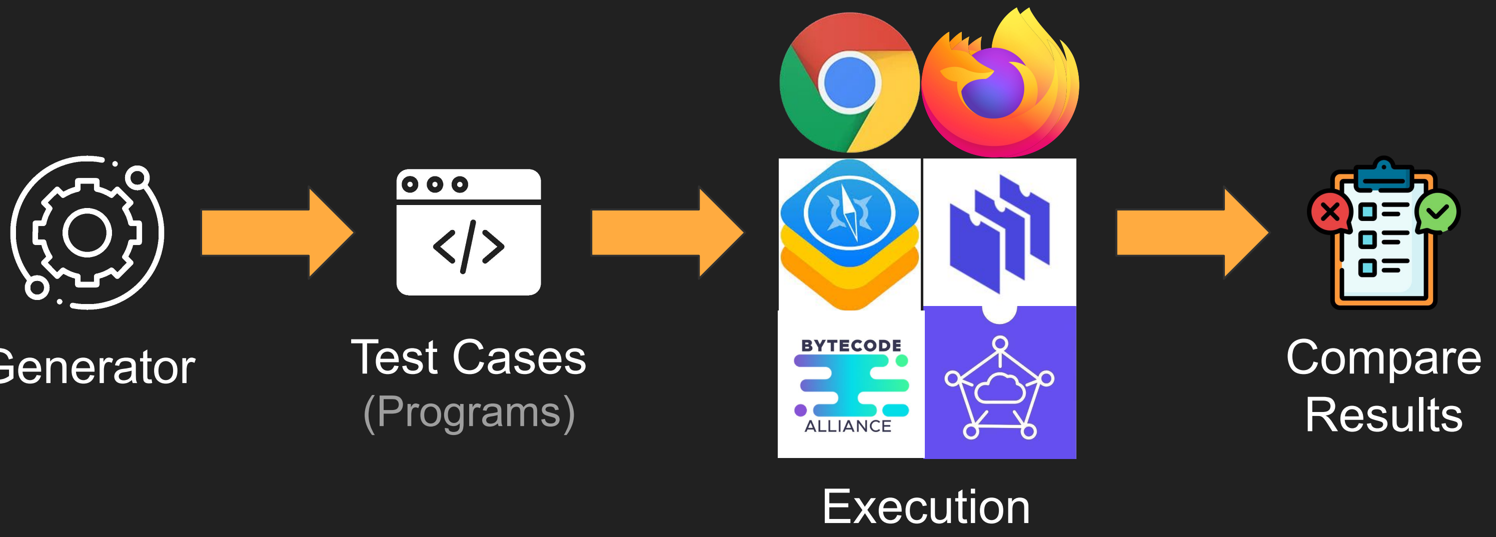
Finding Semantic Bugs

- Differential fuzzing



Finding Semantic Bugs

- Differential fuzzing



Finding Semantic Bugs

- Differential fuzzing

Execution

? How can we generate test cases efficiently?

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

Odds of generating this randomly?
How do we guide this?

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

Odds of generating this randomly?
How do we guide this?

[Preliminary Study]
SOTA fuzzers failed to generate such program
(Xsmith, wasm-smith)

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

❗ Solution: Rule-guided fuzzing
→ Extract the rules and use them in fuzzing

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

Solution: Rule-guided fuzzing
→ Extract the rules and use them in fuzzing

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

[Challenge]
Compiler rules: Defined in IR
Programs: Written in WASM

Approach 1: Rule-guided Fuzzing

- Challenge 1.1: Complex rules

(v0 & v1) | ~v1
→ v0 | ~v1

Optimization

Testing needs:

IR vs. WASM

```
local.get 0
local.get 1
i64.and
local.get 1
i64.const -1
i64.xor
i64.or
```

WASM Program

Solution: Rule-guided fuzzing
→ Extract the rules and use them in fuzzing

[Challenge]
Compiler rules: Defined in IR
Programs: Written in WASM

How do we close the gap??

Approach 1.1: Instruction-level Inference

- Challenge 1.2: Closing the gap between IR and WebAssembly

Instruction-level Inference:

band → i64.and
bor → i64.or
bnot → ???

Approach 1.1: Instruction-level Inference

- Challenge 1.2: Closing the gap between IR and WebAssembly

Instruction-level Inference:

band → i64.and
bor → i64.or
bnot → ???

We do not have a WASM instruction that directly maps to bnot

Approach 1.2: Rule-level Inference

- Challenge 1.2: Closing the gap between IR and WebAssembly

Rule-level Inference:

band → i64.and
bor → i64.or
bnot → *opt. rule**

Refer to other rules
for missing linkages

Approach 1.2: Rule-level Inference

- Challenge 1.2: Closing the gap between IR and WebAssembly

Rule-level Inference:

band → i64.and
bor → i64.or
bnot → *opt. rule**

v3 = iconst.i64 -1
v4 = bxor v1, v3

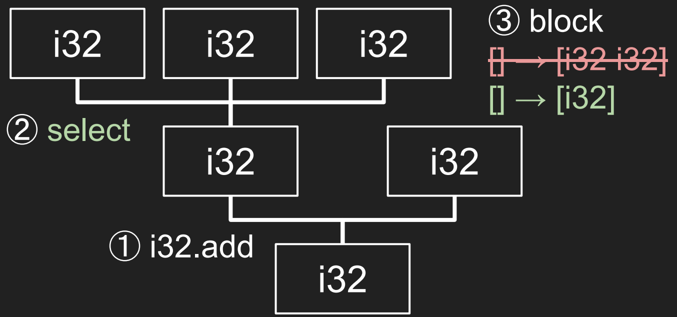
into

v4 = bnot v1

Refer to other rules
for missing linkages

Approach 2: Reverse Stack-based Generation

- Challenge 2: Generate structures or instructions diversely
  - AST-based : limited structure diversity (e.g., blocks)
  - Stack-based: limited instruction diversity (e.g., select)



AST-based

Instructions	***Stack***
① local.get 0	[]
② local.get 1	[i32]
③ i32.add ~~select~~	**[i32 i32]**
④ block	[i32]
[] → [i32 i32]	[i32 i32 i32]

Stack-based

Approach 2: Reverse Stack-based Generation

- **Solution: Reverse stack-based generation**
    - Stack-based generation, but done *reversely*

Approach 2: Reverse Stack-based Generation

- **Solution: Reverse stack-based generation**
  - Stack-based generation, but done *reversely*
  - Observation: Instructions have *only* 0-1 return types
    → Less stack state constraints on generating instructions

Approach 2: Reverse Stack-based Generation

- **Solution: Reverse stack-based generation**
    - Stack-based generation, but done *reversely*
    - Observation: Instructions have *only* 0-1 return types

      → Less stack state constraints on generating instructions

      e.g., v128.bitselect requires 3 v128s on parameters, but only 1 in returns

Evaluation

- Target Runtimes: 6 runtimes
    - wasmtime, Wasmer, WasmEdge, V8, SpiderMonkey, JavaScriptCore
    - Tested various optimization / architectures

- Found 20 new bugs, with one CVE ID (CVE-2023-29548)

Evaluation

- Coverage
    - Able to cover significantly more in wasmtime

Baseline	Coverage	RGFuzz	RGFuzz⁻			wasmtime-differential			Fuzzgen			Wasm-mutate			Xsmith		
		Mean	Mean	Difference	*p*	Mean	Difference	*p*	Mean	Difference	*p*	Mean	Difference	*p*	Mean	Difference	*p*
Optimization	**Line**	**69.37%**	49.29%	20.08%	**0.008**	36.89%	32.48%	**0.008**	46.32%	23.05%	**0.008**	37.85%	31.52%	**0.008**	39.33%	30.04%	**0.008**
	Rule	**70.64%**	51.89%	18.75%	**0.012**	42.79%	27.85%	**0.012**	50.97%	19.67%	**0.008**	40.13%	30.51%	**0.008**	41.02%	29.62%	**0.012**
Lowering	**Line**	**71.43%**	70.54%	0.89%	**0.008**	65.42%	6.01%	**0.008**	56.61%	14.82%	**0.008**	27.22%	44.21%	**0.008**	27.53%	43.90%	**0.008**
	Rule	**75.11%**	74.23%	0.88%	**0.012**	68.73%	6.38%	**0.008**	56.63%	18.48%	**0.008**	27.06%	48.05%	**0.008**	26.47%	48.64%	**0.012**
Total	**Line**	**28.14%**	27.12%	1.02%	**0.008**	17.41%	10.73%	**0.012**	16.46%	11.68%	**0.008**	11.85%	16.29%	**0.012**	20.37%	7.77%	**0.008**

Evaluation

- Coverage
  - Able to cover significantly more in wasmtime
  - Could also efficiently test other runtimes
  - Rule-guided fuzzing was only effective in wasmtime though

Baseline	Coverage	RGFuzz Mean	RGFuzz⁻ Mean	RGFuzz⁻ Difference	RGFuzz⁻ p	wasmtime-differential Mean	wasmtime-differential Difference	wasmtime-differential p	Fuzzgen Mean	Fuzzgen Difference	Fuzzgen p	Wasm-mutate Mean	Wasm-mutate Difference	Wasm-mutate p	Xsmith Mean	Xsmith Difference	Xsmith p
Optimization	**Line**	**69.37%**	49.29%	20.08%	**0.008**	36.89%	32.48%	**0.008**	46.32%	23.05%	**0.008**	37.85%	31.52%	**0.008**	39.33%	30.04%	**0.008**
	Rule	**70.64%**	51.89%	18.75%	**0.012**	42.79%	27.85%	**0.012**	50.97%	19.67%	**0.008**	40.13%	30.51%	**0.008**	41.02%	29.62%	**0.012**
Lowering	**Line**	**71.43%**	70.54%	0.89%	**0.008**	65.42%	6.01%	**0.008**	56.61%	14.82%	**0.008**	27.22%	44.21%	**0.008**	27.53%	43.90%	**0.008**
	Rule	**75.11%**	74.23%	0.88%	**0.012**	68.73%	6.38%	**0.008**	56.63%	18.48%	**0.008**	27.06%	48.05%	**0.008**	26.47%	48.64%	**0.012**
Total	**Line**	**28.14%**	27.12%	1.02%	**0.008**	17.41%	10.73%	**0.012**	16.46%	11.68%	**0.008**	11.85%	16.29%	**0.012**	20.37%	7.77%	**0.008**

Engines	RGFuzz Mean	RGFuzz⁻ Mean	RGFuzz⁻ Difference	RGFuzz⁻ p	Wasm-smith Mean	Wasm-smith Difference	Wasm-smith p	Xsmith Mean	Xsmith Difference	Xsmith p
Wasmer	**12.22%**	12.21%	0.01%	0.234	12.02%	0.19%	**0.012**	11.33%	0.89%	**0.011**
WasmEdge	10.82%	**10.83%**	-0.01%	0.154	10.57%	0.25%	**0.011**	10.37%	0.45%	**0.010**
V8	**19.81%**	19.80%	0.01%	0.264	19.53%	0.28%	**0.011**	17.65%	2.16%	**0.011**
JavaScriptCore	**19.58%**	19.55%	0.03%	0.057	19.07%	0.51%	**0.012**	14.58%	5.00%	**0.012**

Case Study

- Could even cover super complex optimizations

```
(rule (simplify (bor ty @ $I64
    (bor ty
        (bor ty
        (ishl ty x (iconst_u ty 56))
        (ishl ty
            (band ty x (iconst_u ty 0xff00))
            (iconst_u ty 40)))
        (bor ty
        (ishl ty
            (band ty x (iconst_u ty 0xff_0000))
            (iconst_u ty 24))
        (ishl ty
            (band ty x (iconst_u ty 0xff00_0000))
            (iconst_u ty 8))))
    (bor ty
        (bor ty
        (band ty
            (ushr ty x (iconst_u ty 8))
            (iconst_u ty 0xff00_0000))
        (band ty
            (ushr ty x (iconst_u ty 24))
            (iconst_u ty 0xff_0000)))
        (bor ty
        (band ty
            (ushr ty x (iconst_u ty 40))
            (iconst_u ty 0xff00))
        (ushr ty x (iconst_u ty 56))))))
    (bswap ty x))
```

Complex Optimization Rule

```
(rule 6 (lower (shuffle a b (u128_from_immediate
 0x1f0f_1e0e_1d0d_1c0c_1b0b_1a0a_1909_1808)))
         (x64_punpckhbw a b))
```

Specific Immediates

Case Study

- Could find optimization bugs!

```
local.get 0
local.get 1
local.get 1
local.get 0
i64.gt_s
select
;; Expected: min(arg0, arg1)
;; Actual   : max(arg0, arg1)
```

min mistaken as max

wasmtime issue 8114

```
local.get 2 ;; arg2: 0xfff8
f64.load
f64.const 0
local.get 0 ;; arg0: 1
select
;; Expected: mem[arg2] or 0.0 based on arg0
;; Actual   : Trap
```

Load more bytes than expected (as xmm)

wasmtime issue 8112

Key Takeaways

- Two main approaches
  - Rule-guided fuzzing
  - Reverse stack-based generation
- Showed effectiveness in finding optimization bugs