RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes

Junyoung Park
KAIST
School of
Electrical Engineering
parkjuny@kaist.ac.kr

Yunho Kim*
Hanyang University
Department of
Computer Science
yunhokim@hanyang.ac.kr

Insu Yun*
KAIST
School of
Electrical Engineering
insuyun@kaist.ac.kr

***Abstract*—WebAssembly runtimes embed compilers to compile WebAssembly code into machine code for execution. These compilers use various compiler rules to define how to optimize and lower the WebAssembly code. However, existing testing tools struggle to explore these rules effectively due to their complexity. Moreover, they cannot generate test cases diversely due to their limitations, which can result in undetected bugs.**

This paper presents RGFuzz, a differential fuzzer for WebAssembly runtimes, addressing the existing limitations through two novel techniques. First, RGFuzz uses *rule-guided fuzzing*, which extracts compiler rules from the WebAssembly runtime, wasmtime, and uses them to guide test case generation, thereby effectively exploring complex rules. Second, RGFuzz uses *reverse stack-based generation* to generate test cases diversely. These techniques enable RGFuzz to find bugs effectively in WebAssembly runtimes. We implemented RGFuzz and evaluated it on six engines: wasmtime, Wasmer, WasmEdge, V8, SpiderMonkey, and JavaScriptCore. As a result, RGFuzz found 20 new bugs in these engines, including one bug with a CVE ID issued. Our evaluation demonstrates that RGFuzz outperforms existing fuzzers by utilizing the extracted rules and diversely generating test cases.

1. Introduction

WebAssembly [1] is a novel programming language introduced in 2017 [2], featuring a compact and portable binary format that performs at near-native execution speeds. Since its introduction in 2017, WebAssembly has been supported by all major web browsers, such as Chrome, Firefox, and Safari. Its versatility makes it a portable compilation target for other programming languages, such as C/C++ and Rust. WebAssembly now extends beyond web use to applications like smart contracts [3], [4], edge cloud services [5], [6], [7], [8], and image processing [9].

Various WebAssembly runtimes have been developed to execute WebAssembly. Such runtimes (e.g., wasmtime [10], Wasmer [11], and WasmEdge [12]) optimize and lower WebAssembly code into machine code, which is then executed in the host machine. These runtimes use various compiler rules to achieve this, defining how to compile WebAssembly code into efficient machine code. Notably, WebAssembly runtimes must be correct. Otherwise, they can break the security guarantees of WebAssembly, which requires the code to execute correctly and safely.

To uncover bugs in WebAssembly runtimes, one of the most widely used techniques in existing works is fuzzing [13], [14], [15], [16], [17], [18], [19], [20]. While these works have successfully found bugs, they have two limitations. First, they fail to test complex compiler rules, which require specific sequences of certain instructions to trigger them. As existing fuzzers solely rely on the WebAssembly specification and random generation, it is hard for them to generate test cases to test these rules. Second, these fuzzers often limit the diversity of test cases. They focus on generating correct test cases to avoid rejections during the early validation of WebAssembly. However, they may sacrifice diversity in favor of correctness, limiting effectiveness in finding bugs.

To address the limitations of existing fuzzers, we present **RGFuzz**, the **R**ule **G**uided **Fuzz**er for WebAssembly runtimes. RGFuzz is a differential fuzzer with two novel techniques: *rule-guided fuzzing* and *reverse stack-based generation*. RGFuzz uses compiler rules in one WebAssembly runtime, wasmtime, and generates test cases based on the rules. This allows RGFuzz to effectively test the runtimes by generating test cases that can trigger the rules. Moreover, RGFuzz employs a new test case generation method, reverse-stack based generation, to generate correct and diverse test cases. This also improves the effectiveness of differential fuzzing, making RGFuzz more effective in finding bugs in WebAssembly runtimes.

Unfortunately, using the compiler rules in WebAssembly runtimes for fuzzing is not straightforward. Thanks to the recent trends in a WebAssembly runtime, like wasmtime, we can easily extract compiler rules in WebAssembly runtimes. These runtimes define compiler rules using the machine-friendly Domain-Specific Language (DSL). However, these rules translate and optimize their Intermediate Representations (IRs), not WebAssembly instructions. As a result, we need to know how WebAssembly instructions are translated to IRs to use these rules in RGFuzz.

To address this issue, RGFuzz learns the mapping between WebAssembly instructions and IRs in two

* *Co-corresponding authors*

steps: *instruction-level inference* and *recursive substitution*. RGFUZZ first compiles each WebAssembly instruction to IRs to infer their mapping. However, this is not enough as many IRs are not directly converted from WebAssembly instructions but from other IRs. Thus, as its second step, RGFUZZ recursively substitutes the IRs with the corresponding rules to extend the mapping to these IRs.

RGFUZZ also introduces a new test case generation method called reverse stack-based generation. Reverse stack-based generation tracks the stack types while generating WebAssembly instructions, allowing the correctness of the generated test cases. This way also enables RGFUZZ to improve its diversity. Our key insight is that WebAssembly instructions have only zero or one return type, which is more straightforward to satisfy than parameter types. Therefore, if we generate instructions in the reverse order, we can generate more diverse instructions without considering complex stack constraints for their parameters.

We implemented the prototype of RGFUZZ and evaluated it on six engines: wasmtime [10], Wasmer [11], WasmEdge [12], V8 [21], SpiderMonkey [22], and JavaScriptCore [23]. As a result, RGFUZZ found 20 new bugs in these engines, including one bug with a CVE ID issued. RGFUZZ could find these bugs effectively using the extracted rules and diversely generating test cases.

The contributions of our work are the following:

- We present **RGFUZZ**, a differential fuzzer for WebAssembly runtimes with two novel techniques: rule-guided fuzzing and reverse stack-based generation.
- We evaluate RGFUZZ on six WebAssembly engines and demonstrate its effectiveness. Notably, RGFUZZ could discover 20 new bugs in these engines, including one bug with a CVE ID issued.
- We open-source RGFUZZ to foster further research in this area: https://github.com/kaist-hacking/RGFuzz

2. Background

2.1. WebAssembly Runtimes

To evaluate WebAssembly, we require WebAssembly runtime, a software that can execute WebAssembly programs like wasmtime [10], Wasmer [11], or runtimes for browsers (e.g., SpiderMonkey [22] or V8 [21]). This subsection will briefly introduce how WebAssembly runtimes work and why they are important.

Workflow. WebAssembly runtimes compile WebAssembly programs into machine code to provide near-native speed. Since WebAssembly is a low-level, assembly-like language, it can be easily compiled into machine code. For example, the function in Figure 1 directly translates into a single multiplication instruction with a constant of two. To further boost the speed, the runtimes can apply more aggressive optimizations. For example, Figure 1 can be compiled into multiplication initially, but it can be further optimized into addition (`arg0+arg0`) if we enable high optimization levels.

```
(func $mul2 (param i32) (result i32)
            ;; stack: []
local.get 0 ;; stack: [arg0]
i32.const 2 ;; stack: [arg0, 2]
i32.mul     ;; stack: [arg0 * 2]
;; mul2(arg0) => arg0 * 2
```

Figure 1: An example of a WebAssembly function that multiplies an argument by two.

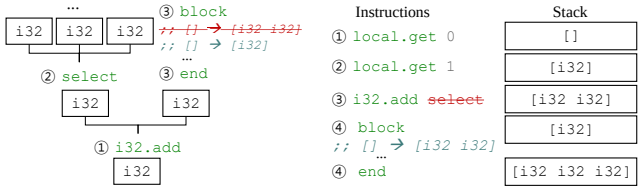
Safety. WebAssembly provides a thoroughly secure execution environment for each instance by offering isolation [24], [25]. Thanks to its strong security, WebAssembly is being applied to various security-critical applications, such as smart contracts [3] or distributed systems [6]. To realize this, WebAssembly implements 1) a strict type system, 2) memory safety through separation and boundary checks, and 3) control flow integrity. First, WebAssembly employs a strict type system validated at compile time. Before compiling modules, the runtime performs type and sanity checks. If a module fails these checks, it is rejected and not compiled. Second, WebAssembly ensures memory safety through separation and boundary checks. It uses linear memory regions isolated from call stacks, locals, and the runtime stack. All memory accesses use offsets, preventing arbitrary memory access via raw pointers. Programs are also restricted from accessing memory outside predefined boundaries. Finally, WebAssembly maintains control flow integrity. Function calls and branches must target valid destinations, and indirect calls undergo runtime type signature checks, trapping on failure. The call stack is isolated from the runtime stack to protect return addresses from overflows.

Stack types. WebAssembly defines *stack types*, a sequence of types that describe how instructions manipulate the runtime stack. These types are defined as `[`t_1^*`] -> [`t_2^*`]`, where t_1^* and t_2^* are type sequences. For example, `[i32 i64] -> [f32 f64]` describes a stack type that takes two values of `i32` and `i64` from the top of the stack and returns two values of `f32` and `f64` to the stack.

WebAssembly enforces conditions on the stack types in specific instructions during validation. For example, WebAssembly requires the same changes in the stack types for the `block` instruction. If a block has a stack type of `[] -> [i64 i64]`, the stack should result in the same state regardless of how the block is executed (e.g., through conditional branches or loops). WebAssembly runtime validates this condition before execution. As a result, if this condition is not met, the module is considered invalid and rejected early.

2.2. WebAssembly Runtime Fuzzing

One of the most promising ways to test WebAssembly runtimes is fuzzing. In fuzzing, it is crucial to generate valid test cases that can pass the validation stage of the runtime. To generate valid test cases, existing works have adopted two main approaches: AST (Abstract Syntax Tree)-based generation and stack-based generation.



(a) AST-based generation that restricts diverse stack types.

(b) Stack-based generation that restricts diverse instructions.

Figure 2: Existing approaches to generate WebAssembly test cases and their limitations.

AST-based generation. AST-based generation, used in many fuzzers like Xsmith [20], generates test cases recursively starting from the return type of a function. This is the same as the ordinary grammar-based fuzzing [26], [27], [28]. In particular, this kind of fuzzer first checks the type on the AST and generates the corresponding instruction that can result in the specific type. It repeats this process until no more leaf nodes are in the AST or the maximum depth is reached. For example, Figure 2a shows how AST-based generation works when the return type of a function is `i32`. In this case, the fuzzer randomly generates an instruction that returns `i32`, such as `i32.add`. After generating this instruction, the fuzzer adds its parameter types as child nodes (e.g., two `i32` for `i32.add`), repeating this process on the children until it finishes generation. This approach is intuitive but limited in handling diverse stack types (see Section 3.3).

Stack-based generation. A new approach called stack-based generation has been proposed to support diverse stack types. Stack-based generation, used in fuzzers like Wasm-smith [17], generates test cases while tracking the stack states. Whenever this type of fuzzer generates a new instruction, it checks the current stack state and decides which instructions can be generated based on the current stack. This is useful to satisfy the stack type constraints in WebAssembly, as it tracks the stack while generating test cases. For example, as shown in Figure 2b, if it finds two `i32` values at the top of the stack, the fuzzer can generate an `i32.add` instruction. After that, it pops two `i32` values and pushes the result of the addition, which is `i32`, back to the stack. By repeating this, the fuzzer can generate a valid test case that conforms to the WebAssembly specification. This is good for handling diverse stack types. However, unlike AST-based generation, it limits the flexibility of generating test cases, making it difficult to generate diverse instructions (see Section 3.3).

2.3. Cranelift ISLE

Cranelift ISLE (**I**nstruction **S**election **L**owering **E**xpressions) [29] is a domain-specific language that defines optimization and instruction lowering rules in the Cranelift compiler of wasmtime [10]. It is designed to aid developers define compiler components more naturally, while enabling machine tools to interpret the semantics of the rules more effectively. ISLE defines these rules as *rewriting rules* [30], written in an S-expression syntax with a left-hand side

```
;; (x ^ -1) can be replaced with the `bnot` instruction
(rule (simplify (bxor ty x (iconst ty k)))
  (if-let -1 (i64_sextend_imm64 ty k))
  (bnot ty x))
```

(a) An optimization rule for `bnot`.

```
;; `or(and(x, y), not(y)) == or(x, not(y))`
(rule (simplify (bor ty
                     (band ty x y)
                     z @ (bnot ty y)))
      (bor ty x z))
```

(b) A complex optimization rule for `bor`.

```
local.get 0  ;; x
local.get 1  ;; y
i64.and      ;; band x y
local.get 1  ;; y
i64.const -1 ;; -1
i64.xor      ;; xor y -1 -> bnot y
i64.or       ;; bor
;; (bor (band x y) (bnot y)) -> (bor x (bnot y))
```

(c) Minimal WebAssembly code to trigger the rule in Figure 3b.

Figure 3: Examples of rules written in Cranelift ISLE.

(LHS) and a right-hand side (RHS). ISLE rules match expressions with patterns on LHS and rewrite them with RHS expressions. Optimization rules are defined as rewriting rules between Intermediate Representation (IR) expressions, while instruction lowering rules define rewriting rules from IR to architecture-specific low-level IR.

Figure 3a shows an example of an ISLE optimization rule, which rewrites `x xor -1` into a `bnot` expression. To describe optimization, ISLE uses a top-level term `simplify` that takes an expression as its argument. The LHS of the example rule is `(bxor ty x (iconst ty k))`, defining a pattern that matches a `xor` operation with `x` and a constant of `k`, where `x`, `k` being the arguments and `ty` being the type of the expression. After matching expressions to the LHS, they are rewritten into the expression in the RHS.

ISLE rules may include *rule conditions* that constrain the expressions using `if-let` clauses, which define conditions with LHS and RHS. To utilize the rule, IRs must satisfy the $LHS = RHS$ condition. Figure 3a contains an `if-let` clause, which checks if the sign-extended value of k to 64-bit equals the value of -1. Due to the condition, the rule only allows an argument `k` to be the value equal to -1.

ISLE rules also define *directives*, which are internal expressions that are used to seamlessly connect ISLE rules to the other compiler components. These directives may define operations that cannot be easily expressed in ISLE. For example, `i64_sextend_imm64` in Figure 3a defines a sign-extension operation of `k` to the type of `i64`. More examples include `fits_in_64`, which limits the type of expression to the integer types of bit-width lesser or equal to 64.

3. Challenges & Our Approaches

In this section, we will describe the technical challenges and our approaches to address these challenges.

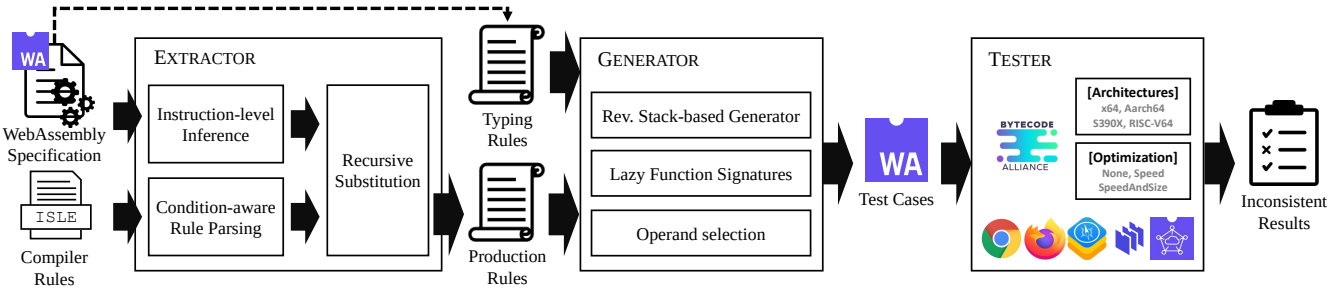


Figure 4: Overview of RGFuzz.

3.1. Complex compiler rules

Challenge. Current WebAssembly fuzzers are not effective in exploring complex compiler rules. This is because they only consider the WebAssembly specification [31] and do not consider the actual runtime implementation. For example, wasmtime has an optimization rule like Figure 3b with three IRs: `bor`, `band`, and `bnot`. To trigger the rule, the test case must contain three IRs with their correct positions and arguments. According to our preliminary study, existing fuzzers, Wasm-smith [17] and Xsmith [20], fail to generate a test case for this rule within 24 hours. To test this rule, a fuzzer needs to exactly generate a sequence of four instructions: `i64.and`, `i64.const -1`, `i64.xor`, and `i64.or`, including three auxiliary `local.get` instructions that indicate the correct arguments (see, Figure 3c). This is too challenging for traditional fuzzers that only rely on the WebAssembly specification, which primarily consists of typing rules.

Solution. To effectively explore these complex compiler rules, our system RGFuzz employs a technique called *rule-guided fuzzing*. In particular, RGFuzz directly utilizes these rules in the WebAssembly runtime wasmtime, written in the Cranelift ISLE, for fuzzing. Previously, WebAssembly runtimes tightly coupled these rules with their implementations, written in ordinary languages (e.g., C, C++), making it difficult to extract the rules as these languages are challenging to analyze. Meanwhile, ISLE introduces a new opportunity; it defines the rules in its machine-friendly language, making it easy to extract the rules and understand their semantics.

3.2. Difficulty in mapping WebAssembly and IR

Challenge. Even after we decide to use the Cranelift rules for fuzzing, this is not trivial because these rules are written in Cranelift IRs, but we need to generate test cases in WebAssembly. It might be possible to directly use IRs for fuzzing (e.g., Fuzzgen [16]). However, it has limitations in terms of portability and coverage; such an IR-based fuzzer cannot be easily adapted to fuzz other runtimes and cannot cover bugs related to WebAssembly to IR translation.

Matching IR to WebAssembly is not always straightforward. Figure 3b shows an example of why this could be challenging. In this example, we might easily guess that `band` and `bor` in IR will be matched with `i64.and` and `i64.or` in WebAssembly, respectively. However, this direct matching is not always possible; `bnot` in IR is not directly matched with any WebAssembly instructions because WebAssembly has no instruction like `i64.not`. To understand this IR, we need to understand another rule in Figure 3a, which rewrites `x xor -1` to `bnot`. This demonstrates that we must understand multiple rules to fully understand their mapping.

Solution. To match IR with WebAssembly, RGFuzz uses *instruction-level inference* and *recursive substitution*. First, RGFuzz infers the direct matching between WebAssembly and IR. For that, RGFuzz iteratively compiles every WebAssembly into IR and finds the mapping between them. Unfortunately, this is insufficient for certain IRs, like `bnot` in Figure 3b. Second, to support such cases, RGFuzz repeatedly substitutes other rules to build more mappings. For example, RGFuzz can substitute the rule in Figure 3a to build `bnot` from `i64.const -1` and `i64.xor`.

3.3. Non-diverse test case generation

Challenge. We observed that existing WebAssembly runtime fuzzers are ineffective in generating diverse test cases. In particular, we found that both AST-based and stack-based program generations have their limitations.

AST-based generation: Non-diverse structures. AST-based program generation limits producing diverse structures (e.g., `block` or `loop`) due to its insufficient support for stack types. Notably, the AST-based fuzzer can only generate instruction sequences with a single return type (e.g., `i32`). This prevents the fuzzer from generating diverse control flow structures like `blocks` or `loops`, which can have multiple types of returns and parameters simultaneously. In fact, we observed that Xsmith, an AST-based fuzzer, generates a control-flow structure with a pre-defined template (e.g., `block (result i32)`), which is less flexible than the one defined in the WebAssembly specification.

Stack-based generation: Non-diverse instructions. Stack-based generation tracks the stack states, allowing it to generate more diverse structures; however, it generates certain instructions much less frequently than others. This is because these instructions require specific stack states that are less likely to happen in a random generation. For example, `v128.bitselect` instruction requires three operands to be located on top of the stack, and their types should be all

the same as `v128`. Unfortunately, this is extremely rare to be satisfied by a random generation without any guidance. According to our evaluation (Section 6.3), Wasm-smith [17], which employs stack-based generation, could only generate only 28 `v128.bitselect` instructions among 100k test cases.

Solution. To address these limitations, RGFUZZ suggests *reverse stack-based generation* to generate diverse test cases. This approach combines the advantages of both AST-based and stack-based generation. In particular, RGFUZZ generates test cases from the return types of the functions like AST-based generation, while tracking the stack states like stack-based generation does. This allows RGFUZZ to generate test cases to have more diverse structures and instructions.

4. Design

4.1. Overview

Figure 4 illustrates the high-level view of RGFUZZ. It has three main components: 1) EXTRACTOR, 2) GENERATOR, and 3) TESTER. Initially, EXTRACTOR extracts optimization and lowering rules from the Cranelift compiler. To understand the relationships between the WebAssembly instructions and the IRs, EXTRACTOR uses instruction-level inference and recursive substitution. Then, it translates these rules into production rules that describe which WebAssembly instructions can trigger the corresponding compiler rules. Next, GENERATOR produces WebAssembly modules using the rules acquired from EXTRACTOR and the typing rules from the WebAssembly specification. To generate diverse test cases, GENERATOR adopts reverse stack-based generation, carefully determining the function signatures and operands. Finally, TESTER runs these modules in the WebAssembly runtimes with diverse architectures and optimizations. TESTER compares the results of multiple executions to identify any mismatches that indicate potential runtime bugs.

4.2. Rule-guided fuzzing

RGFUZZ converts the Cranelift ISLE rules into production rules for fuzzing through a two-step process: 1) instruction-level inference and 2) recursive substitution.

4.2.1. Preprocessing: Condition-aware rule parsing. To build production rules, RGFUZZ first needs to parse the ISLE rules. These rules are complicated and contain many directives and conditional expressions. We need to handle these directives and conditions properly to fully understand the semantics of the rules. In the following, we discuss how RGFUZZ preprocesses ISLE rules.

Directives. To handle directives in the ISLE rules (e.g., `i64_sextend_imm64` in Figure 3a), RGFUZZ embeds a manually written handler for each directive, which converts them into conditions on the operands. These conditions will be used in the matching and test case generation processes. For instance, `i64_sextend_imm64` is to sign-extend the operand to a 64-bit integer. Then, RGFUZZ's handler will mark this condition on the operand. After that, when RGFUZZ generates a test case, it generates a random operand while satisfying this condition.

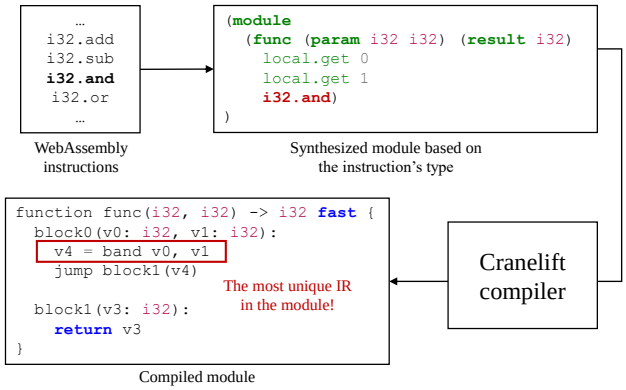


Figure 5: RGFUZZ's instruction-level inference to infer a direct mapping between a WebAssembly instruction and an IR.

if-let clause. `if-let` clauses in ISLE rules are handled similarly to the directives. As we mentioned before, the `if-let` clause describes the condition that should be satisfied to apply the rule. For example, in Figure 3a, this optimization rule is only applied when the constant k equals -1 after the sign extension. RGFUZZ also tracks this condition and uses it to further process. Currently, RGFUZZ only supports simple `if-let` clauses, such as binary operations with constants.

4.2.2. Instruction-level inference. Then, RGFUZZ infers a direct mapping between a WebAsembly instruction and an IR. For example, in this step, RGFUZZ learns that the WebAssembly instructions, `i64.and` and `i64.or` correspond to the IRs `band`, and `bor`, respectively (Figure 3b).

To this end, RGFUZZ iterates over all WebAssembly instructions and infers corresponding IRs, as illustrated in Figure 5. In more detail, initially, RGFUZZ wraps each WebAssembly instruction with a WebAssembly module based on its parameter and return types. This is required as the Cranelift compiler accepts only a module as an input. Then, RGFUZZ translates the WebAssembly module into an IR module using the Cranelift compiler. Unfortunately, the compiled module contains many non-canonical IRs, which are not directly related to the WebAssembly instruction of interest. For example, as shown in Figure 5, the compiled module for `i32.and` contains block-related IRs and control flow IRs, which are not directly related to the instruction. These non-canonical IRs could be more diverse according to the types of WebAssembly instructions. For example, the `i32.eq` instruction in WebAssembly will contain the unsigned extension IRs (i.e., `uextend.i32`) as the Cranelift compiler returns the comparison results as 8-bit integers (i.e., `icmp eq`), which are then extended to 32-bit integers.

To filter out mappings related to non-canonical IRs, RGFUZZ uses a heuristics method that analyzes all compiled IRs and chooses the most unique ones as canonical. In more detail, it first filters out the IRs related to control flow (e.g.,

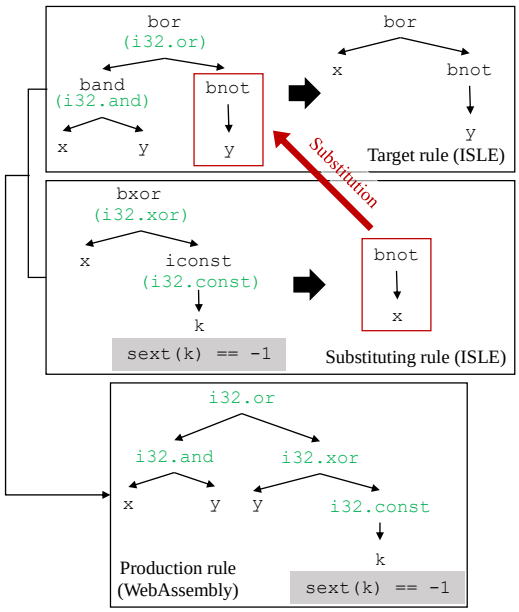


Figure 6: RGFuzz's recursive substitution for Figure 3b. The green text indicates WebAssembly instructions from the instruction-level inference, and the grey box indicates the condition from the condition-aware rule parsing. Even though the rule contains non-trivial IRs like `bnot`, RGFuzz can convert the ISLE rule into a production rule by substituting other rules (marked with red boxes).

```
# Global rng         : a random number generator
# Input  func        : the current function on generation
# Input  rets        : return types of this block
# Input  depth       : a depth of the current block
# Input  targetStack: a desired stack after generation
# Output block       : a generated WebAssembly block
def genBlock(func, rets, depth=0, targetStack=None):
  stack = rets
  instrs = []
  while not rng.terminate():
    if rng.genStruct() and depth < MAX_DEPTH:
      instrs, stack =\
        genStruct(func, stack, instrs, depth)
      continue

    if not stack.isEmpty() and rng.consumeStack():
      newType = stack.pop()
    else:
      newType = None

    if newType is None or rng.useRule():
      rule = rng.choice(getRules(newType))
      stack += rule.params
      instrs += genOperands(func, rule.instrs)
    else:
      if rng.genConst(newType):
        instrs.append(genConstInstr(newType))
      elif rng.genFuncArg(newType):
        arg = func.allocOrReuseArg(newType)
        instrs.append(genLocalGet(arg))

  if targetStack is not None:
    stack, instrs = \
      massageTypes(func, stack, instrs, targetStack)
    assert stack == targetStack

  return Block(instrs, rets, params=stack)
```

Figure 7: The algorithm of reverse stack-based test case generation.

`blockN`, `jump`, and `return`) since they are not directly related to the WebAssembly instructions. Then, RGFuzz prioritizes the mappings that contain the unique IRs, focusing on the instructions that translate to the smallest number of IRs. For example, `uextend` IRs, generated by both extension and comparison instructions, are more closely related to extension instructions. The extension instructions translate into fewer IRs since they only contain `uextend` IRs, while comparison instructions translate into both `uextend` (i.e., for type conversion) and comparison IRs. Therefore, RGFuzz excludes the comparison instructions from the mapping following the heuristics.

4.2.3. Recursive substitution. Next, RGFuzz modifies the ISLE rule into production rules for fuzzing. These production rules express the LHS (Left Hand Side) of the ISLE rule in WebAssembly, allowing it to trigger this specific ISLE rule with a higher probability than a random generation.

This process is divided into two stages: *instruction-level substitution* and *rule-level substitution*. In instruction-level substitution, RGFuzz uses the inference results from the previous step (§4.2.2) to substitute IRs in the ISLE rule with corresponding WebAssembly instructions. For example, in Figure 6, the IR `band` is substituted with the WebAssembly instruction `i64.and`. Unfortunately, this is not enough to convert the ISLE rule into production rules due to the presence of non-trivial IRs like `bnot`. To address this issue, RGFuzz uses the rule-level substitution. In this step, RGFuzz iterates over other ISLE rules and attempts to identify ISLE rules that contain the non-trivial IR in their RHS. Then, RGFuzz uses pattern matching to check if the rule can be substituted with the target rule. If this is the case, RGFuzz substitutes the non-trivial IR with the LHS of the target rule. For example, in Figure 6, the IR `bnot` is replaced with the LHS of the rule in Figure 3a. Finally, RGFuzz converts the ISLE rule into production rules by repeating this process until all non-trivial IRs are replaced with WebAssembly instructions. Notably, RGFuzz maintains the conditions for each operand during this step, which will be used in the test case generation.

RGFuzz also employs a pruning technique to prevent the explosion of the number of production rules during this rule-level substitution. Notably, rule-level substitution can infinitely expand in many cases. For instance, if some rules contain recursion, where an expression appears in both the LHS and RHS of the rule, the rules can be indefinitely expanded if we repeatedly use the rule in the substitution. To mitigate this, RGFuzz substitutes a rule only if it has not been derived from another rule substitution. For example, as shown in Figure 6, the `bor` rule will not be used for substituting other rules since it is derived from the `bnot` rule.

4.3. Reverse stack-based generation

Then, RGFuzz uses reverse stack-based generation to generate test cases. For that, RGFuzz uses typing rules,

```
# rng, func, depth: see genBlock
# Input  stack : a current stack state (reverse)
# Input  instrs: current generated instructions (reverse)
# Output instrs, stack
def genStruct(func, stack, instrs, depth):
  kind = rng.chooseStructKind()
  func.labels.push(kind, stack)
  rets = stack.popN(n=rng.randIndex())

  if kind == BLOCK or kind == LOOP:
    block = genBlock(func, rets, depth+1)
    block.instrs = [End()] + block.instrs \
                   + [Instr(kind)]
  elif kind == IF:
    ifBlock = genBlock(func, rets, depth+1)
    endBlock = genBlock(func, rets, depth+1,
                        ifBlock.params)
    instrs = [End()] + endBlock.instrs + \
             [Else()] + ifBlock.instrs + \
             [If()]
    block = Block(instrs, rets, ifBlock.params)
  elif kind == CALL:
    block = genCall(stack, instrs, rets)
  else: # kind == BR
    brCandidates = func.labels.match(stack)
    if len(brCandidates) == 0:
      return # No label to branch
    brTarget = rng.choice(brCandidates)
    brInstr,brStack = genBr(func, stack, brTarget)
    instrs.append(brInstr)
    stack = brStack

  instrs += block.instrs
  stack += block.params
  return instrs, stack
```

Figure 8: The algorithm of control structure generation.

which define types of WebAssembly instructions from the specification [31], with the production rules.

Reverse stack-based generation. Figure 7 shows the generation algorithm of RGFuzz. As shown in the algorithm, RGFuzz tracks `stack` while in generation, but *reversely* unlike other stack-based generators. At line 8, RGFuzz initializes `stack` with the return types. Then, RGFuzz repeatedly generates instructions until the termination condition is met (Line 10). In particular, RGFuzz can generate a control structure (e.g., block, loop, if), arbitrary instructions based on rules, constant instructions, or local variable instructions (Lines 11–14, 21–24, 26–27, 28–30, respectively). RGFuzz's structure generation algorithm will be explained in the next part. Notably, RGFuzz pops the return types of the instructions or structures (Line 17) and pushes their parameter types back to the stack (Line 23). This is reversed from the existing stack-based generators, which pops the parameters and pushes the returns to the stack.

The key intuition of our approach is that the constraints of the instructions are easier to satisfy if we deal with their return types instead of the parameter types. In particular, in WebAssembly, all instructions have only 0 or 1 return type, allowing us to easily match instructions to the current stack states based on their return types. However, the existing stack-based method focuses on satisfying parameter types, which are more complex and challenging. For example, as we mentioned before, the `v128.bitselect` instruction requires generating three `v128` operands at the top of the stack, which is extremely unlikely to happen in the random generation.

RGFuzz performs *type massaging* to adjust the stack state if we have a specific stack state to satisfy (`targetStack` in Line 34). For example, the `if` structure requires the stack state to be the same at both ends of the `if` block and the `else` block. To satisfy this, RGFuzz sets the target stack state of the `else` block with the `if` block's type.

Type massaging works by popping non-matching types from the stack and pushing missing types. First, RGFuzz pops non-matching types from the stack using arguments and constants, as they have types of `[] -> [t]` for `t` being their type. When these instructions are generated, one type is popped from the stack while no type is pushed back, as RGFuzz builds the stack reversely. Second, RGFuzz pushes missing types with memory stores, which have types of `[i32 t] -> []`. While nothing is popped from the stack because there is no return type, type `t` is pushed to the stack.

Lazy function signatures. Unlike other differential fuzzers for WebAssembly [15], [18], [20], [32], RGFuzz lazily determines its function signature; it generates the function body first and then determines the function signature based on the generated body. In differential testing, the return value of a function is represented as a relation among its function arguments (e.g., `mul2(x) = x*2`). If we define the function signature before generating the function body, we should replace any value with a constant if the given arguments cannot represent it. For instance, in a `select` instruction, whose third operand must always be of type `i32`, if our function arguments do not include i32, this operand must be expressed as a constant. However, this can lead to constant propagation, making us unable to explore more complex optimizations for the `select` instruction. To prevent this, RGFuzz dynamically allocates variables while generating instructions and then determines the function signature based on this (Line 29 in Figure 7).

Generation of control structures. Unlike AST-based generators, RGFuzz can generate control structures (e.g., block, loop, if) diversely and correctly by tracking stack states. Figure 8 shows the algorithm for generating control structures. First, RGFuzz decides which kind of structure to generate (Line 6) and pushes the current state to the labels (Line 7). This label is used later as the target of branch instructions (Lines 24–27). Largely, RGFuzz generates five kinds of structures: `block`, `loop`, `if`, `br`, and `call`. For `block`, `loop`, and `if`, RGFuzz generates a block structure by recursively calling the block generation (Lines 10 – 13). RGFuzz also generates the `call` structure with the function call generation (Line 23). RGFuzz can generate `br` structure if there is a proper label to branch (Lines 28 – 31). Whenever RGFuzz generates these structures, it tracks the stack state by popping n return types from the stack and pushing the parameters of the generated structure back to the stack (Lines 8 and 34). Such parameters are decided based on the final stack type of the generation. With this, RGFuzz can generate structures with diverse block types by dynamically deciding their types without putting constraints on them.

Operand selection. GENERATOR produces random test

TABLE 1: List of runtimes and their versions. The table also lists LLVM backend versions if the runtime uses LLVM.

Runtimes	Version	LLVM-backend
wasmtime	v18.0.1	-
Wasmer	v4.2.6	v15.0.7
WasmEdge	v0.13.5	v16.0.6
V8	v12.6.21	-
SpiderMonkey	c7df16ff	-
JavaScriptCore	cdf0e8ad	-

cases based on the production rules to find bugs hidden behind the complex compiler rules. This allows RGFuzz to discover these bugs more effectively, usually located near the rules. Similar to other standard fuzzers [33], [34], Generator generates instruction operands by choosing new values or from pre-defined interesting value sets. The interesting values contain values that are widely used in compilers. For example, the interesting values for integers contain 0, -1, 1, `SINT_MIN`, and `UINT_MAX`, and those for floats contain 0.0, 1.0, -1.0, `+inf`, `NaN`.

Avoiding non-determinism. RGFuzz must avoid non-determinism in the generated test cases to avoid false positives in differential testing. WebAssembly can be non-deterministic due to two reasons: 1) `grow` instructions such as `memory.grow` can be non-deterministic, and 2) `NaN` floating values can have either positive or negative signs. To avoid this, RGFuzz does not generate `grow` instructions and canonicalize `NaN` values. For `NaN` canonicalization, RGFuzz follows the method used in Wasm-smith, which inserts a sequence of instructions to canonicalize `NaN` values after each instruction that may produce `NaN` values [35]. These instructions check if the result of the target instruction is `NaN` by comparing it with itself (i.e., using `NaN != NaN`) and return a canonicalized value if it is `NaN`.

4.4. Differential testing

RGFuzz loads and runs the modules in different WebAssembly runtimes with varying compilers, optimizations, and architectures. When RGFuzz loads the modules, it generates arguments based on the target function argument types, runs the function multiple times with varying arguments, and compares the results. RGFuzz also randomizes CPU options (e.g., SSE, AVX) before instantiating the module, which allows RGFuzz to explore lowering rules that require specific CPU features. To compare divergence in memory, RGFuzz also hashes the memory states after execution and compares hashes between the runs.

5. Implementation

We implemented RGFuzz with 13.5k lines of Rust and Python. Specifically, we implemented Extractor with 7.7k lines of Rust, Generator with 3.9k lines of Python, and Tester with 1.9k lines of Python. To implement Extractor, we utilized two components from the Cranelift compiler: the ISLE compiler and the code translator. We reused the ISLE compiler to parse ISLE files and the code translator to translate Cranelift IR modules into WebAssembly modules. We also used wasm-ast [36] to build WebAssembly modules. In Generator, we used wasmfun [37] to implement WebAssembly module building and encoding, while modifying it to make it support SIMD instructions.

TABLE 2: List of baseline tools used in the evaluation.

Baseline	Version	Target
wasmtime-differential	v18.0.1	wasmtime
Fuzzgen	v18.0.1	wasmtime
Wasm-mutate	v18.0.1	wasmtime
Wasm-smith	v1.206.0	Generic
Xsmith	792c7695	Generic

Targets. Table 4 in Appendix lists supported targets, architectures, and compiler optimization levels. Currently, RGFuzz supports six engines: wasmtime [10], Wasmer [11], WasmEdge [12], V8 [21], SpiderMonkey [22], and JavaScriptCore [23]. For V8 and SpiderMonkey, RGFuzz uses their architecture simulators, allowing test cases to run with minimal performance losses. For the other engines, RGFuzz uses the latest version of QEMU [38] (v8.2.94).

6. Evaluation

Environments. All experiments were done on servers running Ubuntu 22.04 with two of Intel Xeon Gold 6248R 24-core CPUs and 256GB of RAM. We allocated a single core and 8GB of memory for each fuzzing instance, running them for 24 hours, five times each. For statistical significance, we used Mann-Whitney U tests as proposed by Klees et al. [39].

We performed experiments on six runtimes, listed in Table 1: wasmtime [10], Wasmer [11], WasmEdge [12], V8 [21], SpiderMonkey [22], and JavaScriptCore [23]. We chose their versions as the latest when this paper was written. As Wasmer and WasmEdge used LLVM as their backends, we built LLVM with coverage support to measure the holistic coverage of the runtimes.

Research Questions. In our evaluation, we conducted experiments to answer the following research questions.

- **RQ1.** How effective is RGFuzz in testing a Cranelift-based runtime (i.e., wasmtime) compared to other baselines? (Section 6.1)
- **RQ2.** How effective is RGFuzz in testing other runtimes? (Section 6.2)
- **RQ3.** Can RGFuzz generate test cases diversely? (Section 6.3 and Section 6.4)
- **RQ4.** Can RGFuzz find previously unknown bugs in WebAssembly runtimes? (Section 6.5)

Additionally, we evaluate the validity of the heuristics used in the instruction-level inference in Section A of Appendix.

6.1. Effectiveness in testing wasmtime

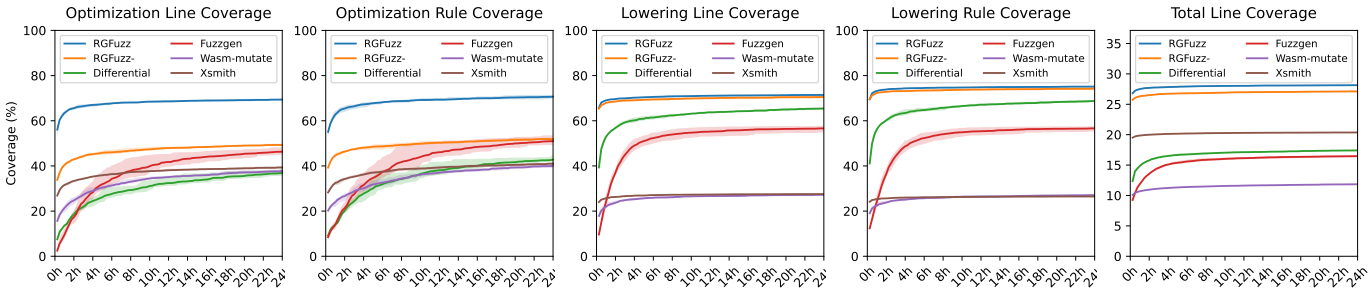


Figure 9: Line and rule coverage measured in wasmtime

```
(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))
```

(a) Example optimization rule requiring the complex IR structure.

```
(rule (lower (has_type $I64 (bswap src)))
    (x64_bswap $I64 src))
```

(b) Example lowering rule requiring an IR from Figure 10a.

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

(c) Example lowering rule requiring specific immediates to trigger the rule.

Figure 10: Examples of rules that RGFuzz was able to cover.

Experimental Setup. To evaluate the effectiveness of RGFuzz in testing wasmtime, a Cranelift-based runtime, we compared it with the baseline tools: wasmtime-differential [15], Fuzzgen [16], Wasm-mutate [14], and Xsmith [20] in Table 2. We also considered Wasm-smith [17] and WADIFF [18]. However, we excluded them because 1) wasmtime-differential internally uses Wasm-smith, and 2) WADIFF's source code was undergoing the tidying-up process at the time of this writing. For the first three baseline tools (wasmtime-differential, Fuzzgen, Wasm-mutate), we used their default configurations by following instructions in their documentation. For Xsmith, we integrated the tool to RGFuzz along with its harness since it cannot run fuzzing in a loop by itself. Moreover, to better show the effectiveness of our rule-guided fuzzing, we also evaluated RGFuzz$^-$, a variant of RGFuzz that only uses the WebAssembly typing rules without production rules from Cranelift ISLE rules.

To compare each tool, we measured their coverage (i.e., total coverage) over 24 hours. For an in-depth comparison, we also measured coverage for each compilation step: optimization and lowering. We focused on the wasmtime files responsible for optimization and lowering rules: `isle_opt.rs` and `isle_x64.rs`. We measured their line coverage and rule coverage. Line coverage, as usual, is the percentage of lines covered by the generated test cases. Rule coverage is the number of optimization and lowering rules the generated test cases cover. This metric directly shows how well each tool explores the compiler rules by the number of rules covered.

Results. Figure 9 shows the line and rule coverage that RGFuzz and the baselines achieved in compiler optimization and lowering code. In summary, RGFuzz shows significantly higher coverage than other baselines in all coverage metrics. In optimization coverage, RGFuzz achieves 69.37% and 70.64% for line and rule coverage, respectively, achieving 23.05% and 19.67% more than the best baseline, Fuzzgen. Meanwhile, in lowering coverage, RGFuzz achieves 71.43% and 75.11% for line and rule coverage, respectively, achieving 6.01% and 6.38% more than the best baseline, wasmtime-differential. Table 5 of Appendix shows the detailed number of the results. Also, Section B of Appendix shows the performance of Extractor.

Impact of rule-guided fuzzing. To understand why RGFuzz outperforms the other baselines, we performed a detailed analysis of the cases where RGFuzz could cover while the baselines could not. After our investigation, we found that RGFuzz could outperform the other baselines because it could cover 1) rules with complex structures and 2) rules requiring specific immediates that are hard to generate without the knowledge of the compiler rules. First, thanks to RGFuzz's rule extraction, it could cover complex rules that require sophisticated constraints on the IR structures. `bswap` is an example of such a rule, which requires a complex IR structure with 35 IRs and specific integer constants (see Figure 10a). As we can easily guess, a naive fuzzer cannot cover such a rule without knowing the

structures. On the other hand, RGFuzz could cover the rule by extracting the rules from the compiler. Second, RGFuzz could cover rules even with specific immediates that are hard to generate randomly. Figure 10c shows a such example, which requires a specific `u128` immediate to trigger the rule (i.e., `0x1f0f_1e0e_1d0d_1c0c_1b0b_1a0a_1909_1808`). As an ordinary fuzzer generates immediate based on random values or a small set of interesting ones, it is impossible to cover such rules if this constant is not in the dictionary. However, since RGFuzz analyzes the compiler rules and extracts specific immediates from the rules, RGFuzz was able to cover such rules.

Ablation Study. To understand the impact of the rule-based approach in RGFuzz, we compared it with RGFuzz$^-$, a variant without rule extraction. Our results show that RGFuzz achieved over 20% higher coverage in optimization and about 0.8% higher coverage in lowering compared to RGFuzz$^-$, showing the effectiveness of our rule-based approach. As previously discussed, RGFuzz could cover complex rules that require knowledge of them to cover.

RGFuzz outperforms RGFuzz$^-$ more significantly in optimization than in lowering because the optimization rules in wasmtime tend to have more complex semantics than the lowering rules. While optimization rules require complex IR structures, lowering rules tend to have simpler structures, mostly not requiring more than three IRs.

6.2. Effectiveness in testing other runtimes

Experimental Setup. To evaluate the effectiveness of RGFuzz in testing other runtimes that are not based on Cranelift, we ran RGFuzz on four runtimes: Wasmer, WasmEdge, V8, and JavaScriptCore. We also tried to evaluate on SpiderMonkey, but we failed to run it with coverage measurement. We found that SpiderMonkey corrupts its coverage data if it encounters undefined instructions (i.e., `ud2`) generated from WebAssembly compilation, making us unable to measure its coverage. In this evaluation, we used Wasm-smith and Xsmith as the baseline tools because other baseline tools in the previous evaluation (wasmtime-differential, Fuzzgen, Wasm-mutate) are unavailable for these runtimes, as they were designed to target only wasmtime.

Results. Figure 11 shows the results of our evaluation in the other runtimes that are not based on Cranelift. In summary, RGFuzz and RGFuzz$^-$ achieved the highest coverage in all the runtimes, showing the effectiveness of our approach in testing versatile runtimes. Based on our investigation, we believe that this happens because of two reasons: 1) RGFuzz and RGFuzz$^-$ can generate diverse test cases that can explore compiler rules effectively (see Section 6.3 and Section 6.4), and 2) RGFuzz could cover the shared rules between the runtimes and wasmtime using the extracted rules. For example, we found that RGFuzz was the only tool that covered the optimization rule in LLVM backends of Wasmer and WasmEdge: $-x \times -y = x \times y$. Other baselines failed to cover this rule because it is difficult to be generated randomly. However, RGFuzz could easily cover it with the extracted rules because it is shared between LLVM backends and wasmtime. Unfortunately, unlike the case of wasmtime, the impact of our rule-based approach was not significant because other runtimes tend to avoid implementing complex rules to reduce compilation time. Despite this, we still believe that RGFuzz could effectively test other runtimes if equipped with more complex, shared rules with wasmtime. Moreover, our approach is generic enough to be applied to other runtimes with machine-readable compiler rules like Cranelift ISLE. Like the previous evaluation, the detailed results are included in Table 6 of Appendix.

6.3. Instruction-level diversity

Experimental Setup. In this experiment, we measured the diversity of test cases generated by RGFuzz and other baselines. We first sampled 100,000 test cases from each tool and measured the distributions of 1) WebAssembly instructions and 2) translated Cranelift IRs. In this evaluation, we used the same baseline tools as the previous experiments: Wasm-smith [17] and Xsmith [20].

To compare the diversity, we measured the cumulative frequency with the *IQR (Interquartile range) filtering* [40]. IQR filtering is the standard method of filtering outliers. We found that some instructions are generated excessively due to the nature of the generation process. For example, `local.get` instructions are always included in every test case for getting function arguments, making the distribution highly biased. Therefore, we used IQR filtering to remove them. We also include the distribution of all instructions (i.e., no filtering) for readers interested in the result involving outliers.

Results. As shown in Figure 12, RGFuzz could generate more diverse test cases than the other baselines as RGFuzz's line is closer to the $y = x$ line than the others. Notably, if a tool generates instructions more diversely than others, the graph will be closer to the $y = x$ line. Theoretically, if one tool generates all instructions with the same frequency, the graph will reach the $y = x$ line. This is because of RGFuzz's reverse-stack generation method. As discussed in Section 4.3, RGFuzz selects instructions reversely from the returns. This makes RGFuzz choose any instructions regardless of their constraints. Unlike RGFuzz, Wasm-smith uses a stack-based generation method, which makes it hard to generate instructions with multiple parameters. For example, we found that Wasm-smith rarely generates `v128.bitselect` instruction, which requires three `v128` parameters. Because it is unlikely to have three `v128` parameters on the top of the stack in a random generation, Wasm-smith could only generate 28 times (frequency of 9.37×10^{-7}) in 100,000 test cases. This demonstrates that the Wasm-smith's stack-based generation has a limitation in generating diverse instructions, while RGFuzz's reverse-stack generation can resolve that.

Even though Xsmith can freely select instructions like RGFuzz thanks to its reverse generation method, it has implementation issues in selecting instructions. As a result, Xsmith fails to generate as diverse instructions as RGFuzz. More specifically, when Xsmith selects instructions, it first

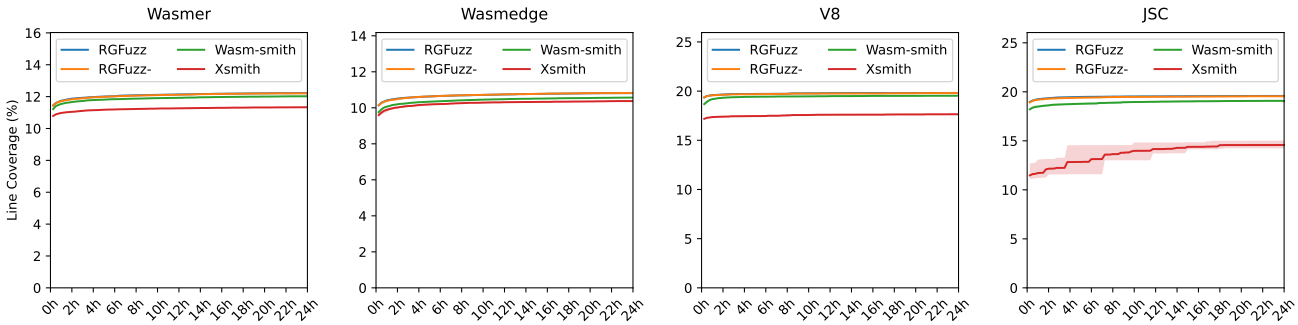


Figure 11: Total line coverage measured in other runtimes.

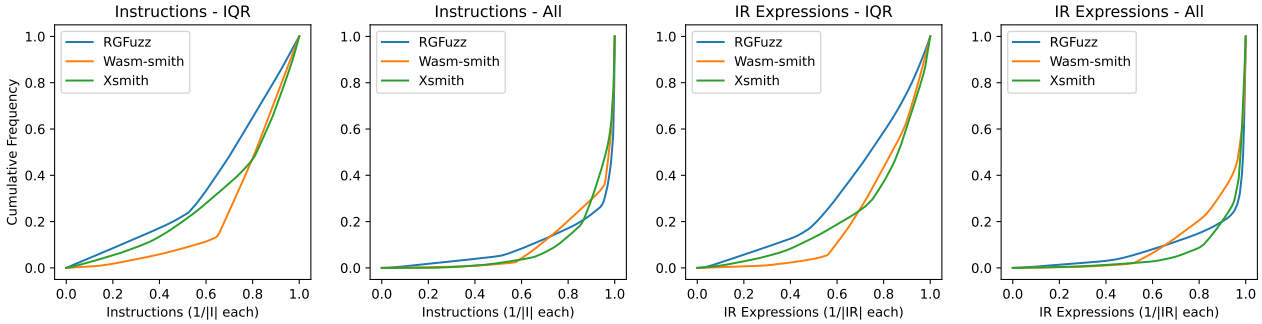


Figure 12: Cumulative instruction and IR frequency (IQR, All).

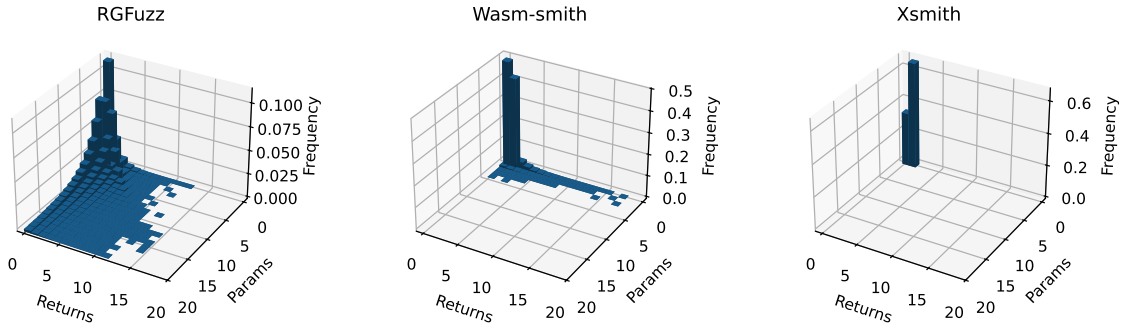


Figure 13: Block type distribution for tools.

chooses a category of instructions (e.g., binary operations, comparisons, etc.). Then, it selects a specific instruction (e.g., `i32.mul` from binary operations) from the category. Unfortunately, Xsmith defines the probability of selecting a category without considering the number of instructions in each category. This causes Xsmith to select a category with fewer instructions more frequently, making its generation less diverse. For example, Xsmith generates `i32.mul` 21,390 times (frequency of 4.21×10^{-3}) in 100,000 test cases, while it generates `i32.ne` 2,670 times (5.26×10^{-4}).

6.4. Structure-level diversity

Experimental Setup. In this experiment, we measured the structure-level diversity from the distribution of block types in the generated test cases. We acquired the block types of `if`, `loop`, and `block` instructions, which are the most common block types in WebAssembly modules. We categorized the types of blocks with the numbers of parameter and return types and measured their frequency in the test cases.

Results. Figure 13 shows the block type distributions of the generated test cases by RGFuzz and the baselines. The graph clearly shows that RGFuzz generates block types more diversely than the others. Specifically, Wasm-smith and Xsmith generate blocks with no parameters and 0 or 1 return type over 90% of the time, while RGFuzz generates blocks with more diverse parameter and return types.

While RGFuzz can generate block types diversely by dynamically generating them, Wasm-smith cannot. Wasm-smith tries to generate blocks with pre-generated block types, and if it fails, it chooses block types between no parameter type and 0 or 1 return type as they do not need

TABLE 3: Discovered bugs by RGFuzz.

#	Runtime	Bug ID	Architecture	State	Description
1	SpiderMonkey	1822754	ARM64	Fixed	Wrong lowering of `udiv` with -2^n
2	SpiderMonkey	1836710	MIPS64	Duplicate	No support of unaligned memory accesses in architecture simulator
3	SpiderMonkey	1836708	MIPS64, MIPS32, Loong64	Fixed	Register allocation bug of `i64.mul`
4	V8	40272422	MIPS64	Fixed	Missing sign-extensions in architecture simulator
5	V8	42204055	RISC-V64	Reported	Program non-termination for `i64.atomic.rmw.xxx`
6	V8	42204056	RISC-V32	Fixed	Bad memory reads with `i64.atomic.rmw.cmpxchg`
7	V8	42204054	RISC-V32	Fixed	Possible memory leak for `i64.atomic.rmw32.xxx_u`
8	V8	42204057	RISC-V32	Fixed	Missing implementation for `i64.atomic.rmw.xchg`
9	V8	40270498	MIPS64, RISC-V64, Loong64	Reported	Missing sign-extensions in architecture simulator
10	V8	40915736	MIPS64, RISC-V64	Fixed	Inconsistent sign extension for `urem` + `extend`
11	V8	40270499	RISC-V32	Reported	Wrong code generation for `i64.lt_s`
12	wasmtime	8114	RISC-V64	Fixed	Wrong lowering for `smin`
13	wasmtime	8112	X64	Fixed	`select` + `load` floating point wrong lowering
14	wasmtime	8132	S390X	Fixed	Wrong endianness for spilled function arguments
15	wasmtime	8145	X64, ARM64, S390X	Fixed	Missing NaN canonicalization for `demote`
16	wasmtime	8131	RISC-V64	Fixed	Vector length bug in `bitselect` + `bitcast` + `cmp`
17	wasmtime	8179	X64, ARM64, S390X	Fixed	Missing NaN canonicalization for `copysign` + `demote`
18	wasmtime	8216	ARM64, RISC-V64	Duplicate	Incorrect memory side-effect for stores near bounds
19	Wasmer	4567	X64	Reported	Inconsistent memory side-effect with traps
20	Wasmer	4568	X64	Reported	Incorrect result for `extmul` instructions
21	WasmEdge	3346	X64	Reported	Wrong optimization for invalid loads
22	WasmEdge	3347	X64	Reported	Inconsistent memory side-effect with traps

to be pre-generated. This results in biases towards these two block types, making the block type distribution less diverse. Moreover, unlike RGFuzz being stack-aware, Xsmith cannot generate block types with multiple return and parameter types. This is the limitation of Xsmith's AST-based generation: such block types cannot be used since stack types cannot be checked properly in the generation process.

6.5. Discovered bugs by RGFuzz

To evaluate whether RGFuzz can find new bugs in WebAssembly runtimes, we ran RGFuzz over 15 months. We intermittently ran RGFuzz on our targets to find bugs while developing RGFuzz. Our longest fuzzing campaign lasted 24 hours, and all new bugs were discovered within that time. As a result, RGFuzz found 22 bugs in the runtimes, with 20 of them previously unknown. We reported all bugs to the vendors. 13 were confirmed and fixed, 2 were duplicates, and 7 are still under investigation. Notably, one confirmed bug in SpiderMonkey was assigned a CVE ID of CVE-2023-29548. While running RGFuzz, we also found a bug in the TCG accelerator of QEMU [38] and another in WasmEdge Rust SDK. These bugs were also reported to the vendors and are confirmed and fixed.

Most of the bugs that RGFuzz found were semantic bugs, leading to incorrect behaviors in the compiled WebAssembly programs. These bugs could be exploited by attackers to induce unexpected behaviors of security-critical WebAssembly programs. For example, we found that wasmtime maintainers consider mis-compilation bugs as security bugs [41]. In the next section (Section 7), we will provide case studies of the bugs found by RGFuzz.

We can estimate the effectiveness of RGFuzz in finding bugs by considering how well our targets have been tested or fuzzed before. For example, wasmtime, in which we found 6 new bugs (excluding 1 duplicate), has been continuously fuzzed with oss-fuzz [42] starting from January 2020[1], before its near-initial release (v0.12.0). Since oss-fuzz includes all the fuzzers we compared against, we believe RGFuzz is capable of finding bugs that current fuzzers cannot. We also note that while the wasmtime developers are actively improving their fuzzers, the bugs remained undiscovered until RGFuzz identified them, demonstrating the effectiveness of RGFuzz.

7. Case Studies

This section presents case studies of the bugs found by RGFuzz in WebAssembly runtimes, discussing how RGFuzz found the bugs while others could not. Here, we present two cases: wasmtime ID 8114 and 8112. Additional case studies are presented in Section C of Appendix.

wasmtime ID 8114. RGFuzz found a bug in wasmtime RISC-V64 that incorrectly lowers `smin` IR into a `max` instruction. Figure 14 shows the proof of concept code of the bug. Because of wasmtime's optimization rule, the WebAssembly instructions in Figure 14 will be converted into a `smin` IR. But, in lowering, the old wasmtime incorrectly lowers this into a `max` instruction, causing a semantic difference from the original WebAssembly instructions.

RGFuzz could find this bug as it learns the optimization rule of `smin` from the compiler. We believe this bug would be hard to be found by other fuzzers since it requires six instructions with specific arguments to trigger the rule.

wasmtime ID 8112. RGFuzz found a bug in wasmtime X64 that incorrectly lowers `select` instruction combined with the `load` of a floating point value. The bug results in

1. https://bugs.chromium.org/p/oss-fuzz/issues/list?q=wasmtime&can=1&sort=-reported

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

Figure 14: Proof of concept code of wasmtime Issue 8114.

a trap with an out-of-bounds memory access even though the memory access is in bounds. Figure 15 shows the proof of concept code of the bug. The code returns a `f64` value loaded from the memory address `arg2` if `arg0` is non-zero. The problem occurs when wasmtime internally handles the `select` as a conditional move but with a `load` into an `xmm` register, which accesses 16 bytes of memory instead of 8 bytes for `f64`. Therefore, when the offset is 8 bytes from the memory max offset (0x10000 - 8), the code results in a trap due to out-of-bounds memory access (0x10000 - 8 + 16).

RGFuzz could find this bug since it can generate instructions diversely even though the instructions require complex constraints. In particular, `select` is rarely generated by other fuzzers, as it requires 1) the first argument to be an `i32` type and 2) the second and third arguments to be the same type. RGFuzz can satisfy this constraint easily due to reverse stack-based generation. However, even though wasmtime has been continuously fuzzed with wasmtime-differential, this bug was undetected since it rarely generated `select` instructions due to its lack of diversity in instruction generation.

```
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
```

Figure 15: Proof of concept code of wasmtime Issue 8112.

8. Discussion and Limitation

Limitations. Our current implementation of RGFuzz focuses on the official WebAssembly specification [31] and the *threads* proposal [43]. This is because we would like to focus on the official specification where all runtimes commonly support. However, some runtimes (e.g., wasmtime) decided to implement other proposals like *relaxed-simd* and *multi-memory*, which RGFuzz does not support. We leave supporting other proposals as future work.

Currently, RGFuzz supports black-box fuzzing with generation, not coverage-guided fuzzing. To implement coverage-guided fuzzing, we need substantial effort to implement feedback mechanisms for the runtimes RGFuzz currently supports. Thus, we leave this as a future work.

Additionally, rule-guided fuzzing, one technique of RGFuzz, showed its limited effectiveness in testing other runtimes (Section 6.2). However, the results of RGFuzz finding 15 bugs in additional runtimes still show that RGFuzz is effective not only for wasmtime but also for other runtimes. This is due to RGFuzz using not only rule-guided fuzzing but also reverse stack-based generation, which showed its effectiveness in Section 6.2. As a result, RGFuzz outperformed existing fuzzers like Xsmith and Wasm-smith across all other runtimes as shown in Figure 11.

Finally, we emphasize the significance of effectively testing Cranelift. Cranelift is not exclusive to wasmtime and is increasingly being adopted in other applications. For example, a widely-used WebAssembly runtime, Wasmer, also supports Cranelift as its compiler backend. Moreover, Cranelift is now being used in a code generator for the Rust compiler [44]. We believe that Cranelift will widen its application in the future.

Compiler rules in a domain-specific language. We believe that maintaining compiler rules separately in a definitive domain-specific language like in wasmtime promotes the development of compiler testers, effectively finding bugs in compilers. With these, the overall reliability and robustness of compilers can be significantly improved. The testers can benefit from this approach by extracting and understanding the semantics of compiler rules.

Several works, including ours, took advantage of the approach to test compilers. RGFuzz was able to extract the semantics of compiler rules from the Cranelift compiler, largely because these rules are written in a definitive and machine-friendly language, ISLE. Similarly, Crocus [45] uses an annotation language based on ISLE to verify ARM64 instruction lowering rules. The work would not be possible without the definitive nature of ISLE.

Manual effort to manage directive handlers. Our directive handlers consist of 2,143 lines of Rust code, managing 274 directives, created over two weeks by someone with limited knowledge of the runtime internals. Although writing directives requires a considerable amount of effort, we want to emphasize that this is a one-time effort.

Handlers may need to be implemented as new directives are added to wasmtime with each major version update introducing around 10 new directives roughly every month. Writing handlers for these new directives takes about 3 hours by reusing existing code, thanks to the similarity among directives. For example, new directives focused on floating-point arithmetics for immediates (e.g., `f32_add/sub/mul/div imm1 imm2`), do not require much effort despite their number due to their consistent pattern. Additionally, even if certain directives are not supported, RGFuzz can still handle all other rules that do not use those specific directives. Therefore, in summary, we believe this level of effort is manageable.

9. Related Work

WebAssembly runtime fuzzing. There have been several studies on fuzzing WebAssembly runtimes [32], [46], [47], [48], [49], [50]. First, WebAssembly runtimes V8 [21] and wasmtime [10] have their own fuzzers [13], [14],

[15], [16] to identify bugs in their implementations. Besides these, Wasm-smith [17] uses stack-based program generation to produce valid WebAssembly modules. In particular, it emulates the runtime stack to ensure the stack conditions of stack-polymorphic instructions (e.g., `loop`). wasmtime-differential [15] employs Wasm-smith and performs differential fuzzing to detect semantics bugs in wasmtime. Fuzzgen [16] is another differential fuzzer in wasmtime but performs IR-based fuzzing. However, due to the nature of IR-based fuzzing, it cannot be used to test other runtimes. WADIFF [18] employs symbolic execution on differential fuzzing, but it can only detect bugs that happen in a single instruction, not the ones that occur with multiple instructions like in optimizations. Perényi et al. [19] uses the stack-directed generator, a backtracking stack-based generator. Similarly to RGFuzz, it generates instructions reversely from the return types while emulating the runtime stack. However, it does not use the stack to handle stack-polymorphism. Instead, it severely limits the polymorphism, generating control structures with only one return and no parameter. Xsmith [20] performs differential fuzzing on runtimes with multiple languages with grammar-based test case generators. Unlike these works, RGFuzz uses rule-guided fuzzing and reverse stack-based generation to effectively find bugs in versatile WebAssembly runtimes.

Compiler Fuzzing. There have been several studies on fuzzing compilers. Csmith [51] is a notable compiler fuzzer that performs differential testing for C compilers, randomly generating test cases while avoiding undefined and unspecified C behaviors. Several works were inspired by Csmith, extending its approach to WebAssembly [17], [20], concurrency [52], OpenCL [53], and Rust [54]. Le et al. proposed Equivalence Modulo Inputs (EMI), generating mutant programs with the same semantics for differential fuzzing. There have been works [53], [55], [56], [57] that succeeded in finding bugs in compilers using EMI. Learning-based approaches [58], [59], [60], [61] that use language models have also been used to generate test cases. More recent works adopt large language models (e.g., GPT [62], [63]) for compiler testing [64], [65].

Differential Testing. Differential testing compares the outputs of two or more implementations of the same specification to find bugs. After McKeeman et al. [66] first proposed the concept of differential testing, it is now widely used to test various applications: browsers [67], [68], [69], [70], [71], [72], compilers [51], [73], [74], virtual machines [75], [76], blockchain [77], [78], and hardware [79]. Similar to these works, RGFuzz adopts the idea of differential testing to find semantic bugs in WebAssembly runtimes.

WebAssembly verification. There have been several studies on verifying WebAssembly runtimes. Crocus [45] verified ARM64 instruction lowering rules in the Cranelift compiler and found two previously unknown bugs. VeriWasm [80] verifies x86_64 binaries and checks if they violate WebAssembly's isolation guarantees. VeriWasm does not verify compilers, but it verifies binaries and inserts safety mechanisms into them. WaVe [81] implements a runtime with verified interactions with the host system. Bosamiya et al. [48] proposes vWASM, a verified sandboxing compiler using formal methods. It also proposes rWASM, which provides safety using safe Rust code. WasmRef-Isabelle [82] implements a monadic interpreter verified with Isabelle/HOL. While these verification works are important for ensuring the correctness of WebAssembly runtimes, fuzzing is still needed as it can be applied to diverse runtimes and can find bugs not covered by the verification.

10. Conclusion

In this paper, we presented RGFuzz, a differential fuzzer for WebAssembly runtimes with two novel techniques: rule-guided fuzzing and reverse stack-based generation. First, RGFuzz uses rule-guided fuzzing, which extracts compiler rules from the WebAssembly runtime and uses them to guide test case generation, effectively exploring complex rules. Second, RGFuzz uses reverse stack-based generation to generate test cases diversely. We implemented the prototype of RGFuzz and evaluated it on six engines: wasmtime, Wasmer, WasmEdge, V8, SpiderMonkey, and JavaScriptCore. As a result, RGFuzz found 20 new bugs in these engines, including one bug with a CVE ID issued.

Acknowledgments

We thank the anonymous reviewers and shepherd for providing valuable feedback. The authors from KAIST were supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2022K1A3A1A91094267) and BK21 FOUR(Connected AI Education & Research Program for Industry and Society Innovation, KAIST EE, No. 4120200113769). The author from Hanyang University was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1A5A1021944), the research fund of Hanyang University(HY-202000000002755), and Samsung Electronics.

References

[1] "Webassembly - roadmap," https://webassembly.org/features/, accessed: 2024-09-20.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.

[3] Block.one, "Eoiso blockchain software & services," https://eos.io/, accessed: 2024-09-20.

[4] ewasm, "Ethereum flavored webassembly (ewasm)," https://github.com/ewasm/design, accessed: 2024-09-20.

[5] Cloudflare, "Webassembly on cloudflare workers," https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, accessed: 2024-09-20.

[6] Fastly, "Compute | fastly products," https://www.fastly.com/products/compute, accessed: 2024-09-20.

[7] Krustlet, "Krustlet," https://krustlet.dev, accessed: 2024-09-20.

[8] wasmCloud, "wasmcloud," https://wasmcloud.com, accessed: 2024-09-20.

[9] S. O'Dwyer, "Photon: A webassembly image processing library," https://silvia-odwyer.github.io/photon/, accessed: 2024-09-20.

[10] B. Alliance, "wasmtime: A standalone runtime for webassembly," https://wasmtime.dev, accessed: 2024-09-20.

[11] Wasmer, "Wasmer," https://wasmer.io, 2024, accessed: 2024-09-20.

[12] WasmEdge, "Wasmedge," https://wasmedge.org, 2024, accessed: 2024-09-20.

[13] Google, "v8/v8/test/fuzzer/wasm-compile.cc," https://chromium.googlesource.com/v8/v8/+/refs/heads/main/test/fuzzer/wasm-compile.cc, accessed: 2024-09-20.

[14] J. C. Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-mutate: Fuzzing webassembly compilers with e-graphs," in *E-Graph Research, Applications, Practices, and Human-factors Symposium*, 2022.

[15] B. Alliance, "fuzz/fuzz_targets/differential.rs," https://github.com/bytecodealliance/wasmtime/blob/main/fuzz/fuzz_targets/differential.rs, accessed: 2024-09-20.

[16] ——, "fuzz/fuzz_targets/cranelift-fuzzgen.rs," https://github.com/bytecodealliance/wasmtime/blob/main/fuzz/fuzz_targets/cranelift-fuzzgen.rs, accessed: 2024-09-20.

[17] ——, "wasm-smith: A webassembly test case generator." https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith, accessed: 2024-09-20.

[18] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "Wadiff: A differential testing framework for webassembly runtimes," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Kirchberg, Luxembourg, Sep. 2023.

[19] Á. Perényi and J. Midtgaard, "Stack-driven program generation of webassembly," in *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings 18*. Springer, 2020, pp. 209–230.

[20] W. Hatch, P. Darragh, S. Porncharoenwase, G. Watson, and E. Eide, "Generating conforming programs with xsmith," in *Proceedings of the Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*, Lisbon, Portugal, Oct. 2023.

[21] Google, "V8 javascript engine," https://chromium.googlesource.com/v8/v8.git/, accessed: 2024-09-20.

[22] Mozilla, "Spidermonkey javascript/webassembly engine," https://spidermonkey.dev, accessed: 2024-09-20.

[23] A. Inc., "Javascriptcore - webkit documentation," https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html, accessed: 2024-09-20.

[24] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita *et al.*, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancover, Canada, Mar. 2023.

[25] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening webassembly against spectre," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[26] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Virtual, Jul. 2021.

[27] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, Jun. 2008.

[28] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[29] B. Alliance, "ISLE: Instruction Selection/Lowering Expressions DSL," https://github.com/bytecodealliance/wasmtime/tree/main/cranelift/isle, accessed: 2024-09-20.

[30] M. Bezem, J. W. Klop, and R. de Vrijer, *Term rewriting systems*. Cambridge University Press, 2003.

[31] "WebAssembly Core Specification," https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. [Online]. Available: https://www.w3.org/TR/wasm-core-2/

[32] G. Hamidy, "Differential fuzzing the webassembly," 2020.

[33] M. Zalewski, "american fuzzy lop (2.52b)," https://lcamtuf.coredump.cx/afl/, accessed: 2024-09-20.

[34] L. Project, "libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, accessed: 2024-09-20.

[35] B. Alliance, "crates/wasm-smith/src/core/code_builder.rs," https://github.com/bytecodealliance/wasm-tools/blob/2728f93ab66edda25a865bc54b81f66f98ca344a/crates/wasm-smith/src/core/code_builder.rs#L1340, accessed: 2024-09-20.

[36] M. D. Salcedo, "A rust-native webassembly syntax model useful for generating, parsing, and emitting webassembly code." https://github.com/misalcedo/wasm-ast, accessed: 2024-09-20.

[37] A. Klein, "Getting the hang of wasm - generate wasm from python," https://github.com/almarklein/wasmfun, accessed: 2024-09-20.

[38] F. Bellard, "Qemu, a fast and portable dynamic translator." in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, Jun. 2005.

[39] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[40] J. W. Tukey *et al.*, *Exploratory data analysis*. Springer, 1977, vol. 2.

[41] A. Crichton, "Miscompilation of wasm 'i64x2.shr_s' instruction with constant input on x86_64," https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-gw5p-q8mj-p7gh, accessed: 2024-09-20.

[42] K. Serebryany, "OSS-Fuzz-google's continuous fuzzing service for open source software," 2017.

[43] WebAssembly, "Threads and atomics in webassembly," https://github.com/WebAssembly/threads, accessed: 2024-09-20.

[44] T. R. P. Language, "Cranelift codegen backend for rust," https://github.com/rust-lang/rustc_codegen_cranelift, accessed: 2024-09-20.

[45] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown, "Lightweight, modular verification for webassembly-to-native instruction selection," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Diego, CA, Apr. 2024.

[46] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. Chan, "Wasmfuzzer: A fuzzer for webassembly virtual machines," in *34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022*. KSI Research Inc., 2022, pp. 537–542.

[47] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-oriented Trends Symposium*, 2021, pp. 23–30.

[48] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe multilingual software sandboxing using WebAssembly," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[49] F. Labs, "WARF - WebAssembly Runtimes Fuzzing project," https://github.com/FuzzingLabs/wasm_runtimes_fuzzing, accessed: 2024-09-20.

[50] C. Wen, "Wasmfuzz: Fuzz testing on javascriptcore and webassembly in webkit," https://github.com/wcventure/WasmFuzz, 2019, accessed: 2024-09-20.

[51] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011.

[52] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 187–196, 2013.

[53] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[54] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, Jul. 2023.

[55] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[56] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Amsterdam, Netherlands, Oct.–Nov. 2016.

[57] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers. pacmpl, 1, oopsla (2017), 93: 1–93: 29," 2017.

[58] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, Jul. 2018.

[59] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language Model-GuidedJavaScript engine fuzzer," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[60] H. Xu, Y. Wang, S. Fan, P. Xie, and A. Liu, "Dsmith: Compiler fuzzing through generative deep learning model with attention," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–9.

[61] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1044–1051.

[62] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[63] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[64] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, Apr. 2024.

[65] Q. Gu, "Llm-based code generation method for golang compiler testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2201–2203.

[66] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[67] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[68] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler," in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[69] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2z2: Detecting rendering regressions in web browsers through differential fuzz testing," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, USA, May 2022.

[70] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+ 1-version differential testing of both javascript engines and specification," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, Madrid, Spain, May 2021.

[71] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proceedings of the 2021 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Virtual, Jun. 2021.

[72] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son, "Diffcsp: Finding browser bugs in content security policy enforcement through differential testing," in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2023.

[73] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[74] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancover, Canada, Mar. 2023.

[75] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 2023.

[76] D. Maier, F. Fäßler, and J.-P. Seifert, "Uncovering smart contract vm bugs via differential fuzzing," in *Reversing and Offensive-oriented Trends Symposium*, 2021, pp. 11–22.

[77] S. Kim and S. Hwang, "Etherdiffer: Differential testing on rpc services of ethereum nodes," in *Proceedings of the 18th European Software Engineering Conference (ESEC) / 31st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, California, USA, Dec. 2023.

[78] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Jul. 2018.

[79] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2021.

[80] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust but verify: Sfi safety for native-compiled wasm," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[81] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2023.

[82] C. Watt, M. Trela, P. Lammich, and F. Märkl, "Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly," in *Proceedings of the 2023 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, Jun. 2023.

Appendix A.
Validity of instruction-level inference heuristics

We manually checked the IR mappings for 363 WebAssembly instructions supported by EXTRACTOR and found three types of generated mappings. First, there were 1-1 mappings (93.91%), were all correct when we manually checked them. Second, there were 1-N mappings, which we found both valid (4.43%) and invalid (1.66%) cases. Among them, the valid cases were feasible as IRs may be mapped to multiple instructions; for instance, when there is no IR support for signedness, an IR may be mapped to both signed and unsigned instructions. Finally, there was one case of no mapping: the `drop` instruction, which cannot be mapped to any IR since it does not emit any IR.

Appendix B.
Performance of rule extraction

In this evaluation, we evaluate the performance of RGFUZZ's rule extraction process (i.e., how many rules can be extracted from how many rules in the ISLE). In summary, RGFUZZ's EXTRACTOR successfully extracts 27,610 production rules from 70.97% (264 out of 372) optimization rules and 40.77% (786 out of 1,928) lowering rules. EXTRACTOR outputs more production rules than the number of ISLE rules due to concretization and substitution. In particular, RGFUZZ concretizes generic types in ISLE rules to the concrete types of WebAssembly instructions. For example, in the case of Figure 3b, this rule can be mapped into `i32.or`, `i64.or`, or `v128.or`. Moreover, EXTRACTOR can substitute multiple rules in its recursive substitution. For example, `ineg`, which negates the operand, can be matched with rules such as $0 - x$, $(!x) + 1$, $!(x - 1)$, and $(-1) \times x$. When this kind of IR is used in other rules, RGFUZZ can build multiple rules by substituting these diverse rules for thorough exploration.

RGFUZZ could extract rules from all the ISLE rules except for the ones that contain: 1) unused IRs (e.g., `umulhi`), 2) IRs that can be translated only from the instructions we have not implemented (e.g., relaxed-simd WebAssembly feature), and 3) directives that are unsupported by RGFUZZ. We believe we can improve the coverage of the extracted rules by implementing more WebAssembly instructions and features, but we leave them as future work.

RGFUZZ takes 2-10 minutes to run, depending on processor performance. On our evaluation machine (Section 6), rule extraction had an average runtime of 151.03 seconds and a maximum memory usage of 1.29 gigabytes. These results are based on 5 runs, with standard deviations of 1.28 seconds and 310.81 kilobytes in memory usage.

Appendix C.
Additional Case Studies

SpiderMonkey ID 1822754 (CVE-2023-29548). SpiderMonkey ARM64 Ion compiler incorrectly lowers unsigned division with a divisor constant -2^n. Figure 16 shows the proof of concept code of the bug. In the code, `i32.div_u` uses a constant value -4, which is -2^2, as the divisor. Ion compiler tries to use an optimization that rewrites unsigned division with 2^n with a left-shift operation with n. However, the signedness of the divisor is not considered when checking if the divisor is a power of two. Therefore, the compiler wrongly considers -2^2 as 2^2 and rewrites the divisor into a left-shift operation with 2. This causes the code to be compiled into a machine code with wrong semantics, different from the original unsigned division with -4.

```
local.get 0
i32.const -4 ;; any value of -2^n will work
i32.div_u
;; Expected: arg0 / 0xfffffffc
;; Actual  : arg0 / 4
```

Figure 16: Proof of concept code of SpiderMonkey ID 1822754 (CVE-2023-29548).

wasmtime ID 8132. RGFUZZ found a bug in wasmtime S390X that incorrectly lowers function calls with spilled stack arguments. Figure 17 shows the proof of concept code of the bug. The code contains a function with nine `v128` arguments, where the last one is passed through the stack instead of registers. When spilling the arguments to the stack, compilers should consider the endianness of the values and swap the byte order of vectors if needed. However, wasmtime did not consider the endianness, leading to wrong executions since the arguments are used with wrong endianness.

To find this bug, a fuzzer must generate a function with at least nine `v128` arguments, making at least one vector argument spilled to the stack. RGFUZZ could find this bug since it can generate diverse block structures, unlike the other fuzzers that cannot generate complex ones. As seen in Figure 13, other fuzzers may generate the required block structure, with an extremely low chance of generating such structures having at least nine parameters.

```
(func (;0;) (type 0) (param v128 v128 v128 v128 v128 v128
      v128 v128 v128) (result v128)
    local.get 5
    local.get 8
    i16x8.extmul_high_i8x16_s)
;; Expected: extmul_high(arg5, arg8)
;; Actual  : extmul_high(arg5, byte_rev(arg8))
```

Figure 17: Proof of concept code of wasmtime Issue 8132.

TABLE 4: Supported architectures and runtimes in RGFuzz.

Engines	Compilers	Optimization Levels	Architectures	Emulation
wasmtime	Cranelift	None, Speed, SpeedAndSize	x86_64, ARM64, S390X, RISC-V64	QEMU
Wasmer	Cranelift LLVM	None, Speed, SpeedAndSize None, Less, Default, Aggressive	x86_64	QEMU
WasmEdge	AOT non-AOT	O0, O1, O2, O3, Os, Oz -	x86_64	QEMU
V8	Liftoff, Turbofan	-	x86_64, ARM64, IA32, ARM, Loong64, MIPS64el, S390X, PPC64, RISC-V64, RISC-V32	Built-in
SpiderMonkey	Baseline, Ion	-	x86_64, ARM64, IA32, ARM, Loong64, MIPS64	Built-in
JavaScriptCore	BBQ, OMG	-	x86_64, ARM64	QEMU

TABLE 5: Line and rule coverage measured in wasmtime with RGFuzz and other baselines. All coverage values showed statistical significance ($p < 0.05$) against RGFuzz. As this table shows, RGFuzz achieves significantly higher coverage than the other baselines thanks to our rule-based approach.

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**

TABLE 6: Total line coverage measured in versatile runtimes. All show statistical significance ($p < 0.05$) against RGFuzz. In all engines, RGFuzz shows the highest coverage compared to baselines, showing the capability of RGFuzz testing versatile runtimes.

Engines	RGFuzz	RGFuzz$^-$			Wasm-smith			Xsmith		
	Mean	Mean	Difference	p	Mean	Difference	p	Mean	Difference	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**

Appendix D.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper proposes techniques to improve rule-guided fuzzing for WebAssembly runtimes. The proposed tool, RGFUZZ, leverages compiler rules in Cranelift compiler and a new test case generation method to generate correct and diverse test cases. This leads to the generation of tests that increase the code coverage and as a result 20 new bugs were found on multiple WebAssembly engines.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) The authors promised to make publicly available a tool for fuzzing WebAssembly runtimes. Such a tool could be very valuable, since it could facilitate rigorous evaluation of the WebAssembly ecosystem.
2) This paper takes advantage of rules extracted from Cranelift ISLE to deal with known issues in rule-guided fuzzing for WebAssembly runtimes. The proposed techniques led to the discovery of new bugs on multiple WebAssembly engines.

D.4. Noteworthy Concerns

1) Some of the reviewers raised concerns regarding the generality of the proposed approach. Additional experiments could clarify how general the approach is.

Appendix E.
Response to the Meta-Review

Response to Concern 1. We believe RGFUZZ is still effective in finding bugs in various runtimes as shown by the results of RGFUZZ finding 15 bugs in additional runtimes. Moreover, given that Cranelift, the compiler of wasmtime, is increasingly being adopted in a growing number of applications, we believe effectively testing Cranelift can be considered significant. However, we agree that RGFUZZ showed its limited effectiveness (particularly for rule-guided fuzzing) in testing the runtimes other than wasmtime.