WEST: Specification-Based Test Generation for WebAssembly

Dongjun Youn
School of Computing
KAIST
Daejeon, South Korea
f52985@kaist.ac.kr

Wonho Shin
School of Computing
KAIST
Daejeon, South Korea
new170527@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

***Abstract*—WebAssembly (Wasm) is a low-level binary instruction format designed for safe and high-performance execution across diverse computing environments and runtimes. As Wasm evolves with new features and proposals, testing the correctness and conformance of Wasm runtimes has become increasingly complex. Manually constructing test suites is labor-intensive and difficult to scale, especially as the specification grows in complexity. While fuzzing-based approaches offer partial automation, they often lack a principled connection to the formal specification, and adapting to evolving or restricted subsets of the specification typically requires manual intervention.**

In this paper, we present WEST, a *specification-based* test generation framework that automatically produces Wasm test cases from mechanized specifications written in *SpecTec*, a Wasm-specific specification language. Given any full or subset variant of the Wasm specification as input, WEST aims to systematically generate test programs that conform to the input grammar and validation rules, and capture the runtime behavior defined by its execution semantics. The framework allows flexible integration of different test generation strategies. For instance, we demonstrate both *top-down* and *bottom-up* approaches for generating Wasm modules, but the architecture is compatible with other generation techniques as well. The framework enables the creation of customized test cases for engines that support only subsets of the Wasm specification. We evaluate WEST across multiple specification variants and engine configurations, demonstrating that it produces valid and diverse test cases. As a result, it reveals 16 bugs across four Wasm engine implementations, 11 of which are confirmed and fixed. We believe that this work provides a solid foundation for future specification-driven test generation.

***Index Terms*—WebAssembly, test generation, language specification, mechanized specification**

I. Introduction

WebAssembly (Wasm) [1] is a low-level binary instruction format designed to support safe, efficient, and portable execution of programs across a wide range of computing environments. One of the defining characteristics of Wasm is its rigorously specified design. The language has a precise formal specification [2], which defines its syntax, static validation rules, and runtime semantics based on a stack-based computation model. Originally created for web applications, Wasm is now used in various domains, including cloud platforms [3], edge computing [4], and blockchain systems [5]. Consequently, a variety of Wasm runtimes now exist beyond traditional web browsers, ranging from lightweight embedded engines [6] to high-performance server-side platforms [7].

To ensure that different Wasm implementations behave consistently with the official specification and across various platforms, rigorous testing is essential. Traditionally, test cases have been manually written to cover diverse aspects of the language. Wasm language designers themselves have developed an official conformance test suite [8], which serves as a baseline for engine compliance. While manually crafting tests enables basic conformance checking, the process is inherently labor-intensive and difficult to scale. Each new feature or proposal demands additional effort to write tests and maintain relevant test coverage. Thus, there is growing demand for automated testing techniques that can reduce manual effort while maintaining test quality and coverage. Researchers have actively explored automated testing techniques [9]–[14], including random program generation via fuzzing and symbolic execution, to discover edge cases and identify potential bugs. Tools such as wasm-smith [9] have been instrumental in generating valid Wasm modules for stress-testing implementations.

However, automated test generation for Wasm is non-trivial. The Wasm specification imposes strict static validation rules that must be satisfied for a test to be executable. These constraints become increasingly complex as new features such as SIMD [15], threads [16], tail calls [17], exceptions [18], and garbage collection [19] are introduced. Testing such features in isolation is already challenging, and testing their interactions adds further complexity. Moreover, Wasm engines do not all implement the same set of features: some support the Wasm 3.0 draft [20], some support only Wasm 2.0, and others include various combinations of proposals [21]. This diversity makes it difficult for conventional fuzzers or test generators to remain compatible across engines or to adapt easily as the specification evolves, as they rely on hand-written encodings of the specifications or ad-hoc validation logic, which are hard to maintain and prone to semantic mismatches.

A recent development that opens up new possibilities for addressing these challenges is SpecTec [22]–[24], a domain-specific language (DSL) and tooling framework for mechanizing the Wasm specification. SpecTec provides a DSL for formally expressing the syntax, validation rules, and runtime semantics of Wasm in a machine-readable format. Moreover, SpecTec facilitates working with subset variants [25] of the Wasm specification by enabling or disabling specific proposals

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00119

or feature groups, or extending the specification with new feature proposals. This mechanized approach simplifies the maintenance of the specification and allows developers to automatically generate various artifacts such as formal definitions, prose pseudocode descriptions, and interpreters. In particular, it enables a new paradigm for building test generators—deriving them directly from the specification, instead of re-implementing fragments of it in an ad-hoc manner.

In this paper, we present a specification-driven framework called WEST (**We**bAssembly **S**pecification-based **T**esting), which uses SpecTec to automatically produce valid and diverse test cases for Wasm. Given a Wasm specification written in SpecTec, it automatically generates tests using the syntax and validation rules, and obtains their results according to the runtime semantics. Importantly, the framework works even when the input specification is altered to reflect a subset, extension, or experimental proposal of Wasm, since the framework relies on the specification itself, with minimal hardcoded logic. As a result, WEST serves as a principled framework for producing conformance tests tailored to different Wasm variants, with minimal manual effort. This design allows WEST to support multiple Wasm engines with varying feature sets, and to adapt seamlessly as the language evolves.

The framework supports pluggable strategies for Wasm module generation. As example strategies, it provides both *top-down* generation, which expands programs using grammar's production rules while enforcing validation rules, and *bottom-up* generation, which builds modules around target instruction sequences while satisfying validation constraints. Other generation strategies can also be integrated into the framework, as its modular architecture is agnostic to specific methods of program construction.

We demonstrate the effectiveness of our approach by applying WEST to several versions of the Wasm specification, including the full Wasm 3.0 draft and its subset variants. Across these configurations, WEST successfully generated test cases, uncovering subtle errors across multiple Wasm engines, particularly in newly-introduced language features. Our evaluation reveals 16 bugs across two major web browsers and two popular Wasm runtimes, 11 of which have been confirmed and fixed, showing the practical impact of our approach. We believe that WEST advances the state of the art in Wasm testing and provides a foundation for specification-driven test generation and fuzzing.

The contributions of this paper include the following:

- We propose a framework for specification-driven Wasm test generation based on mechanized input specifications written in SpecTec.
- We implement a WEST prototype, which generates valid test scripts with runtime assertions directly derived from the Wasm runtime semantics.
- We evaluate WEST on multiple specification variants and Wasm runtimes, uncovering 11 previously unknown bugs.

In the remainder of this paper, we first present a brief overview of Wasm and its mechanized specification framework, SpecTec (§II). We then describe the design goals and overall architecture of WEST (§III), followed by two strategies for Wasm module generation: top-down strategy (§IV) and bottom-up strategy (§V). We show the evaluation results of the WEST implementation with its bug detection capability (§VI), discuss related work (§VII), and conclude (§VIII).

II. Background

This section provides background on Wasm and its specification. We introduce the key components of the Wasm specification and explain how SpecTec mechanizes them. We also review grammar-based generation as a basis for our work.

A. Wasm and Its Specification

Wasm supports a virtual machine with strong typing, making it suitable for secure and efficient execution in various environments, including browsers and standalone runtimes. A Wasm *module* is the unit of instantiation and execution for a Wasm program. It is composed of multiple sections such as globals, functions, and memories, each of which may be empty. The following example shows a simple Wasm module:

```
(module
  (global (export "cnt") (global (mut i32)))
  (func (export "inc") (param i32)
    ;; push the argument to the operand stack
    (local.get 0)
    ;; push the global to the operand stack
    (global.get 0)
    ;; pop two values and push their sum
    (i32.add)
    ;; pop one value and set it to the global
    (global.set 0)))
```

This module defines a single mutable global variable `cnt` of type `i32`, and a function `inc` that takes an integer parameter and adds it to the current value of `cnt`. The function body consists of a sequence of four instructions. The computation model for the instructions is based on a stack machine. First, the instructions `local.get 0` and `global.get 0` push two `i32` values to the operand stack, one from the function argument and one from the global variable `cnt`. Then, the `i32.add` instruction pops both values from the stack, computes their sum, and pushes the result back to the stack. Finally, the `global.set 0` instruction pops the result from the stack and stores it back into the global variable.

The Wasm specification consists of three main components:

- **Syntax** defines the abstract structure and the concrete representations of Wasm programs. Each instruction in Wasm corresponds to a unique opcode in the binary format, allowing for compact and efficient representation.
- **Validation rules** specifies the well-formedness conditions that a module must satisfy to be considered valid. These static rules ensure that Wasm programs are safe to execute. Validation is specified relative to a *context*, which collects relevant information about the surrounding module and the definitions in scope.
- **Runtime semantics** describes the runtime behavior of Wasm programs in the small-step operational semantics style. The entry points of the Wasm execution are *module instantiation* and *function invocation*.

These components define a robust and portable execution environment that supports secure and predictable program behavior, regardless of the underlying hardware or host system. The specification is rigorously written in both prose descriptions and formal definitions, which makes Wasm particularly suitable for formal reasoning and mechanization.

B. SpecTec

SpecTec [22] is a specification framework designed to mechanize the Wasm specification and systematically generate various artifacts from it. It supports both the official Wasm 2.0 specification and the ongoing 3.0 draft, making it a practical and forward-compatible tool for engaging with the evolving Wasm standard. In SpecTec, specifications are written in a DSL particularly designed to concisely express conventional paper-and-pencil notation to define programming language syntax and semantics. This DSL allows users to define *relations* and *rules* that capture the semantics of the language, in a style reminiscent of logic programming languages such as Datalog [26]. For example, the following SpecTec code declares a binary relation `LessThan` over pairs of natural numbers and defines its rules:

```
relation LessThan: nat <: nat

rule LessThan/rule1:
  n1 <: n2
  -- if n1 + 1 = n2

rule LessThan/rule2:
  n1 <: n2
  -- LessThan: n1 <: n3
  -- LessThan: n3 <: n2
```

The first rule has a single *premise* beginning with `--`: it states that `n1 <: n2` holds if `n2` is equal to `n1 + 1`. Rules can also be defined in terms of other relations, allowing for compositional and recursive definitions. For instance, the second rule has two recursive premises, stating that `n1 <: n2` holds if there exists an intermediate `n3` such that both `n1 <: n3` and `n3 <: n2` relations hold.

At its core, SpecTec treats formal specifications themselves as structured data. This representation enables automated transformation, analysis, and synthesis. By encoding syntax, validation rules, and execution semantics in a uniform, machine-readable format, SpecTec bridges the gap between formal language definitions and their artifacts. This data-centric perspective not only ensures consistency and maintainability of the specification, but also facilitates the automatic generation of prose pseudocode descriptions, formal definitions, interpreters, machine-checkable proofs, and tests.

C. Wasm Specification in SpecTec

1) Syntax: SpecTec supports definitions of the Wasm language syntax in BNF-like grammar productions. Nonterminal symbols are written in lowercase letters, and terminal symbols in uppercase or non-alphanumerical letters. For example, the following defines the syntax of Wasm instructions:

```
syntax instr = ...
  | CONST numtype num
  | LOAD numtype memidx memarg
  | ...
```

where the nonterminal `instr` can be expanded into a term that starts with specific terminal symbols, such as `CONST` or `LOAD`, followed by sub-terms that can be recursively expanded further.

SpecTec also supports struct-like syntax definitions. For example, the following defines `context`:

```
syntax context = {
  TYPES deftype*,
  FUNCS deftype*,
  GLOBALS globaltype*,
  MEMS memtype*,
  TABLES tabletype*,
  ...
}
```

used in validation rules. The structured syntax allows field-based access, as in `c.GLOBALS`, which denotes a list of `globaltype`s. An element of the list can be accessed using the index notation, as in `c.GLOBALS[0]`.

2) Validation Rules: Wasm validation rules ensure that modules are well-formed. These rules are defined in terms of the context `c`. It contains type information for each section of a module such as types, functions, globals, memories, and tables, and is constructed by collecting relevant information about the module and the definitions in scope. Wasm has a convention that, for each abstract syntax definition, there is a typing rule that specifies the constraints of the syntax. In SpecTec, each nonterminal `sym` has its corresponding validation relation named `sym_ok`. For example, the following defines the validation rule for the definition `instr`:

```
relation Instr_ok:
  context |- instr : instrtype

rule Instr_ok/load-val:
  c |- LOAD nt x memarg : at -> nt
  -- if c.MEMS[x] = at lim PAGE
  -- if $(2^(memarg.ALIGN) <= $size(nt)/8)
```

Intuitively, this rule states that a `LOAD` instruction is valid with its type `at -> nt` under context `c`, if two premises are satisfied: the memory type at index `x` of the context's memory matches the expected type and alignment constraints are met.

3) Runtime Semantics: SpecTec defines the Wasm runtime semantics as small-step reduction rules over instruction sequences. For example, the following defines the relation `Step_read` of the form `state; instr* ~> instr*`:

```
relation Step_read: state; instr* ~> instr*

rule Step_read/load-oob:
  z; (CONST at i) (LOAD nt x memarg) ~> TRAP
  -- if $(i + memarg.OFFSET + $size(nt)/8
      > |$mem(z, x).BYTES|)

rule Step_read/load-val:
  z; (CONST at i) (LOAD nt x memarg) ~> (CONST nt c)
  -- if $nbytes_(nt, c) =
    $mem(z, x).BYTES[i + memarg.OFFSET : $size(nt)/8]
```

which describes how an instruction sequence reduces to another instruction sequence during execution, under a certain state. The first rule `Step_read/load-oob` describes that the `LOAD` instruction reduces to `TRAP` if the memory access is out-of-bound, while the second rule `Step_read/load-val` describes how the `LOAD` instruction produces a value if the memory access is within bounds.

D. Indirect Interpretation via Algorithmic Translation

SpecTec supports *indirect* interpretation of Wasm modules by directly interpreting the Wasm specification itself. As the specification defines how a given Wasm module is interpreted, directly interpreting the semantics amounts to indirectly interpreting the given Wasm module. Because SpecTec defines runtime semantics in a declarative style using inference rules, it translates them into an algorithmic style as in prose descriptions before interpretation. The translated semantics is expressed in a low-level intermediate representation called *Algorithmic Language (AL)*, which describes the operational behavior of Wasm programs in an imperative, step-by-step manner—for example: `1. Push X to the stack`. SpecTec provides an AL interpreter that directly executes each AL instruction, thereby enabling principled measurement of dynamic coverage over the specification itself.

E. Grammar-Based Test Generation

Grammar-based test generation is a well-established approach for automatically producing syntactically correct input data. For program generation, the language grammar is used to recursively construct well-formed programs. Starting from the top-level nonterminal symbol, production rules are applied to expand nonterminal symbols into terminal symbols or other nonterminals, eventually yielding a complete program.

This method provides a systematic way to generate structurally correct programs, and has been widely adopted in compiler testing, fuzzing, and input generation tools. A key advantage is that the syntactic constraints are enforced by construction. However, traditional grammar-based test generation typically focuses solely on syntactic correctness, without taking into account the language's static semantics or runtime behavior. For languages like Wasm, where validation rules impose complex constraints on types, stack usage, and contexts, grammar-based test generation alone is insufficient to guarantee the generation of valid modules. This limitation motivates the need to integrate grammar expansion with additional validation checking, as described in the next section.

III. WEST

This section presents WEST, our framework for specification-based test generation. We describe its design goals and overall architecture, showing how it generates valid Wasm modules using various strategies and constructs executable test cases directly from mechanized specifications.

A. Design Goals

We design WEST with the following design goals:

- The framework should support a wide range of Wasm specifications, including both full and subset variant versions. It should be able to process any variation of the Wasm specification without requiring modifications to the test generation system itself, except for minor heuristics.
- The generated test cases should be syntactically correct and pass the validation rules with respect to the input specification. Ensuring validation correctness is critical to guarantee that the tests are executable.
- The generated test cases should be *semantically rich*, meaning that they exercise diverse and complex runtime behaviors specified by the input specification, including a wide range of instruction usage patterns, control-flow structures, and semantic edge cases.

This design enables WEST to generate various valid test cases tailored to specific features or engine configurations, allowing fine-grained testing of certain features of interest, or testing runtime implementations that support only a subset of the Wasm specification.

Algorithm 1 Generation of Wasm Tests from a Specification

```
1:  function GenTests(spec, n)
2:      tests ← [ ]
3:      for i ← 1 to n do
4:          modules ← GenModules(spec.syn, spec.vrules)
5:          result ← CompResult(spec.sem, modules)
6:          tests ← tests + ConsTest(modules, result)
7:      return tests
8:  function GenModules(syn, vrules)
9:      modules ← [ ]
10:     l ← SampleLen()
11:     while |modules| < l do
12:         if SampleStrategy() = top-down then
13:             candidate ←
                    TopDown.GenModule(syn, vrules)
14:         else
15:             names ← SampleInstrSeq()
16:             candidate ←
                    BottomUp.GenModule(syn, vrules, names)
17:         modules ← modules + candidate
18:     return modules
19: function CompResult(sem, modules)
20:     results ← [ ]
21:     for each module ∈ modules do
22:         inst ← Interp(sem, instantiate, [module])
23:         calls ← [ ]
24:         if inst = OK then
25:             for each f ∈ module.exported_funcs do
26:                 args ← SampleArgs(f)
27:                 calls ← calls + Interp(sem, invoke, [f, args])
28:         results ← results + (inst, calls)
29:     return results
```

B. Test Generation Framework

Algorithm 1 outlines the overall structure of our test generation framework. The main function GenTests (line 1) takes as input a mechanized Wasm specification *spec* written in SpecTec, and a desired number of test cases n. The input specification *spec* contains the formal definitions of Wasm's syntax (*spec.syn*), validation rules (*spec.vrules*), and execution

semantics (*spec.sem*). These components are used directly during test generation: the syntax and validation rules are used to generate valid modules, while the execution semantics enable extracting runtime behavior of generated modules. As a result, the function returns a set of test cases conforming to the specification (line 7), generated through the following three stages:

a) Module Generation: In line 4, the function call to GENMODULES synthesizes one or more valid Wasm modules. It is defined from line 8, taking two inputs: *syn* and *vrules*. It samples a random length of modules to generate (line 10) and yields a sequence of modules of that length, constructed by calls to strategy-specific GENMODULE functions. It randomly selects between two module generation strategies—*top-down* and *bottom-up*—for each iteration (line 12). In brief, the top-down strategy (line 13) begins from the nonterminal `module` and recursively expands sub-nonterminals guided by validation constraints, while the bottom-up strategy (line 15) starts with a randomly chosen instruction sequence and attempts to synthesize a minimal module that is sufficient to embed it. We describe each GENMODULE strategy in more detail in Section IV and Section V. By leveraging both strategies, WEST explores the generated program space in depth and breadth, leading to diverse feature interactions. While we currently use relatively simple random generation and generate each set of modules independently in each loop iteration (lines 3–7), the system is extensible to incorporate symbolic execution or coverage-guided strategies.

b) Behavior Extraction: In line 5, COMPRESULT computes the expected runtime behaviors of the generated modules. It is defined from line 19, taking two inputs: a runtime semantics sem, and a list of generated modules. The output is a list of semantic results, one per module, each consisting of the instantiation outcome and the results of invoking exported functions. COMPRESULT performs *indirect interpretation* of the formal semantics defined in sem. For each module (line 21), the specification-level function `instantiate`[1] is interpreted using INTERP (line 22) to simulate module instantiation. If it succeeds (line 24), for each exported function of the module (line 25), the arguments are randomly sampled, produced by SAMPLEARGS according to the function's type (line 26). Then, the function invocation is simulated by another call to INTERP (line 27), interpreting the specification-level function `invoke` with the target Wasm function and arguments. The results of both instantiation and invocations are collected (line 28) and returned (line 29).

This approach ensures that runtime behaviors are derived solely from the specification, making the framework agnostic to the behavior of a particular engine.

c) Test Construction: In line 6, the framework calls CONSTEST to translate each module and its corresponding result into a test script in the `.wast` format. The test script is self-contained and executable on any Wasm engine supporting the test format. It contains the following assertions to check that the observed behaviors match the expected behaviors:

- `assert_uninstantiable`: Module instantiation fails due to a runtime trap.
- `assert_return`: Function call returns a value.
- `assert_trap`: Function call results in a trap.
- `assert_throw`: An uncaught exception is thrown

[1]Note that a specification-level function is different from a Wasm function. It is defined in the specification and interpreted by the AL interpreter.

Algorithm 2 Top-Down Generation of a Wasm Module

```
1:  function GENMODULE(syn, vrules)
2:      global (syn, vrules) ← (syn, vrules)
3:      return GEN(module, [ ])
4:  function GEN(sym, stack)
5:      if ISPRIMITIVE(sym) then
6:          return GENPRIMITIVE(sym)
7:      if sym = instrs then
8:          return GENINSTRS(stack)
9:      repeat
10:         prod ← CHOOSEPRODUCTION(syn, sym)
11:         prod ← FIXITERLEN(prod)
12:         prems ← EXTRACTPREMS(vrules, prod)
13:         sub_syms ← EXTRACTSYMS(prod)
14:         sub_terms ← [ ]
15:         for each sub_sym ∈ sub_syms do
16:             stack′ ← stack + (prod, sub_terms)
17:             sub_terms ← sub_terms + GEN(sub_sym, stack′)
18:         term ← REPLACESYMS(prod, sub_syms, sub_terms)
19:     until SATISFYPREMS(stack, term, prems)
20:     return term
21: function GENINSTRS(stack)
22:     (begin, end) ← INFERTARGETTYPE(stack)
23:     instrs ← [ ]
24:     l ← SAMPLELEN()
25:     for i ← 1 to l do
26:         repeat
27:             instr ← GEN(instr, stack)
28:         until POPPABLE(instr, begin) ∧
29:                 (i = l ⟹ POP(instr, begin) = end)
30:         begin ← POP(instr, begin)
31:         instrs ← instrs + instr
32:     return instrs
```

IV. TOP-DOWN STRATEGY

This section introduces the top-down generation strategy used in WEST. The top-down generation strategy follows a recursive grammar expansion approach, guided by the syntax and validation rules defined in the specification.

A. Top-Down Module Generation

The overall structure of the top-down generator is described in Algorithm 2. The entry point is GENMODULE (lines 1–3), which globally initializes the input syntax and validation rules, and initiates generation from the top-level nonterminal symbol `module` with the empty context.

The core of the top-down process is the recursive function GEN (lines 4-20), which takes as input a grammar symbol *sym* and a syntactic context stack *stack* containing the parent symbols and already-generated sibling terms, and returns a valid term corresponding to *sym*. If the input symbol is primitive (e.g., `nat`, `bool`), the generator produces a concrete value of that type at random using an auxiliary function GENPRIMITIVE (line 6). If the symbol is `instrs`, GENINSTRS defined from line 21 generates a sequence of Wasm instructions, as will be explained later in this section.

For general nonterminal symbols, CHOOSEPRODUCTION (line 10) retrieves all productions associated with *sym* from the specification and selects one randomly. If the selected production contains iterative constructs (i.e., Kleene stars, such as `t*`), it invokes FIXITERLEN (line 11) to concretize the number of repetitions into a fixed length. For example, a production `syntax functype = t* -> t*` may be instantiated as `syntax functype = t t -> t`. Next, EXTRACTPREMS (line 12) finds the validation rule associated with the current nonterminal symbol from the specification based on the naming convention and returns the set of premises from that rule. Then, EXTRACTSYMS (line 13) finds all nonterminal symbols *sub_syms* on the right-hand side (RHS) of the selected production. These symbols are recursively instantiated to produce concrete subterms. For each symbol in *sub_syms* (line 15), the generator updates the context stack by appending the current production and the subterms generated so far, $(prod, sub\_terms)$ (line 16), which is used for validating terms with context-dependent premises. Each *sub_sym* is passed recursively to GEN to generate its corresponding concrete subterm, accumulated into a list (line 17). Once all subterms *sub_terms* are generated, REPLACESYMS (line 18) substitutes *sub_syms* with them, yielding a fully constructed candidate term *term*. Then, SATISFYPREMS (line 19) checks whether the validation premises *prems* hold for *term* under the current stack context *stack*. For a simple boolean premise, it attempts to evaluate the premise using concrete values from the term and its context. For a relational premise, it recursively applies the corresponding rules from the specification, effectively interpreting the validation logic to determine if the relation holds. In both cases, the premise is considered satisfied only if it evaluates to `true`. If the premises are not satisfied, the generator retries from line 10 with a new production. This process continues until a valid term is successfully generated. In rare cases, it becomes impossible to generate a valid term, and the generation gets stuck. To prevent such cases, WEST applies a configurable retry limit, after which it abandons the current term and restarts generation.

This recursive structure allows the generator to combine syntax-directed expansion with validation rules in a principled way. By requiring each candidate subterm to satisfy the validation rule associated with its nonterminal symbol, the generator incrementally filters out invalid candidates and guides the generation toward valid modules.

Example. `limits` is a Wasm definition describing the bounds of resizable storage for memory and table types. To illustrate how validation is integrated into the top-down generation, consider the following simplified syntax and validation rule for `limits` defined in SpecTec:

```
syntax limits = [ nat .. nat ]
rule Limits_ok:
  C |- [ n .. m ] : OK
  -- if n <= m
```

The generator would use the given production rule and construct a term `[ n .. m ]`, where `n` and `m` are randomly chosen. If it produces `[ 1 .. 2 ]`, for example, the premise is reduced to `1 <= 2`, which evaluates to `true`. Thus, this term is accepted. On the other hand, if the generator constructs `[ 2 .. 1 ]`, the premise is reduced to `2 <= 1`, which evaluates to `false`, causing the generator to discard the term and retry.

The function GENINSTRS (lines 21–32) is a tailored procedure for generating a sequence of Wasm instructions, the validity of which is defined in terms of stack behavior; each instruction consumes and produces value types on an implicit operand stack. It infers the required stack types at the beginning (*begin*) and end (*end*) of an instruction sequence from the context stack *stack* by INFERTARGETTYPE (line 22). It then attempts to generate a random-length (line 24) instruction sequence satisfying the stack types by recursively calling GEN with the grammar symbol `instr` and the current context stack (line 27). The generated candidate instruction *instr* is accepted only if it can be safely type-checked under the current operand stack, determined by the predicate POPPABLE which checks if the operand stack meets the instruction's type requirements (line 28). For the final instruction ($i = l$), an additional check ensures that the resulting stack type matches the expected ending stack type (line 29). Once accepted, the operand stack is updated using POP (line 30), and the instruction is appended to the sequence (line 31). This stack-aware strategy can detect ill-typed instructions in the middle of generating instruction sequences, improving the efficiency of generation by reducing invalid retries.

B. Wasm-Specific Heuristics

While WEST follows a general grammar-based generation strategy, Wasm's unique design and constraints require a range of domain-specific heuristics to produce valid and diverse modules effectively, which are not explicitly reflected in the algorithm. We describe several such heuristics that WEST incorporates to handle practical challenges and improve generation efficiency.

1) Approximation of Typing Context: Validation rules in the Wasm specification often involve a typing context `C` such as `C.GLOBALS[i] = gt` and `C.FUNCS[i] = ft`. The specification formally describes how a context `C` is constructed from a module in a declarative manner. Rather than explicitly constructing `C` as the specification defines, WEST approximates it using the context stack *stack*. As the stack increasingly tracks all the sibling terms, it naturally stores all relevant context information, such as declared globals, function signatures, and local variables. When a validation rule requires looking up a context, WEST looks up the stack. Instead of constructing

Algorithm 3 Bottom-Up Generation of a Wasm Module

```
1: function GenModule(syn, vrules, names)
2:     prems ← [ ]
3:     types ← InferTypes(names)
4:     (instrs, prems) ← InferImmediates(names, types)
5:     (block, prems) ← WrapBlock(instrs, prems)
6:     (func, prems) ← WrapFunc(block, prems)
7:     module ← WrapModule(func, prems)
8:     return module
```

`c` exactly, approximating it just enough to satisfy validation during generation with low overhead balances the faithfulness to the specification with practicality.

2) Deferred Function Generation: In a current sequential generation process, function call instructions can only refer to the previously generated functions, limiting the ability to generate realistic programs such as mutually recursive calls. To support forward references, WEST defers generation of function bodies. It first declares functions with randomly assigned, well-typed signatures, without bodies. Then, it synthesizes the body of each function using pre-declared function signatures, which allows functions to call any functions within a module while preserving type correctness. This decoupling allows WEST to generate more semantically diverse modules.

3) Constant Instruction Generation: In Wasm modules, specific syntactic positions, such as initializers of `global` and `table` sections, require only *constant* instructions, which are a restricted subset of Wasm instructions. WEST automatically recognizes such positions and instructions from the specification, and ensures that only allowed constant instructions are generated for such positions during construction.

4) Subtype Transitivity Resolution: The subtyping rules of Wasm include transitivity: if `t1 <: t3` and `t3 <: t2`, then `t1 <: t2`. Automatically checking transitivity requires finding a valid intermediate type `t3`. Since WEST does not support such an existential reasoning, we manually rewrite relevant validation rules in the input specification. In particular, we desugar transitive subtyping rules by enumerating a finite set of intermediate candidates for `t3` and explicitly expand the rule cases. This approach avoids recursive explosion while ensuring that subtyping remains decidable and tractable during test generation. While this transformation is manual, it is straightforward and needed only once per affected rule.

5) Filtering Administrative Instructions: The Wasm specification includes administrative instructions such as `TRAP` and `LABEL_`, which are used internally during runtime execution but never appear in the surface syntax. WEST excludes these instructions from generation so that the generated modules contain only surface-syntax instructions and can be parsed and executed by the standard Wasm engines.

V. Bottom-Up Strategy

This section introduces the bottom-up generation strategy used in WEST. The bottom-up strategy synthesizes a minimal valid Wasm module that embeds a given instruction sequence. Unlike the top-down approach, which expands from the top-level nonterminal, the bottom-up approach begins with a list of instruction names and builds outward, constructing the necessary surrounding structures to make the instruction sequence valid.

The motivation for adopting a bottom-up strategy is that it can generate rare edge-case instructions more easily than the top-down strategy. For example, the instruction `array.fill` requires two specific conditions: 1) a mutable array type must be defined in the module, and 2) the operand stack must contain four values of specific type in the correct order. Generating a Wasm module that satisfies all conditions on the type and operand stack is non-trivial and unlikely to be achieved through random grammar expansion. In contrast, the bottom-up strategy starts from the instruction and systematically constructs its surrounding type and operand stack, making such generation straightforward.

Algorithm 3 presents the overall structure. GenModule takes three inputs: the syntax, validation rules, and a list of instruction names. It translates *names* to corresponding instructions with fixed operand types and embeds them into a valid execution context through a sequence of transformation steps. Throughout the transformations, it collects premises (*prems*) representing constraints that the validation context must satisfy. Each transformation step may add new premises to *prems* or resolve existing ones by generating appropriate syntactic constructs.

The first step (line 3) infers the instruction types using InferTypes. It takes the input instruction names and resolves their pre- and post-stack types. This step performs type unification to infer instruction types and ensure consistency across the instruction sequence, based on the Wasm specification. For example, given a sequence of `t.const` and `ref.i31`, their types are `[]->t` and `i32->i31ref`, respectively. Because the post-stack type of the former should be the pre-stack type of the latter, InferTypes deduces that `t = i32`. If it is impossible to resolve types for the given sequence, then the system reports a failure. Unconstrained types after this step are instantiated with random types.

In line 4, InferImmediates takes the instruction names and their inferred types, and attempts to instantiate their immediate operands such as constants, indices, and type annotations. It first collects the premises of the instructions' validation rules and tries to resolve them. Simple premises like `t = i32` are resolved immediately, whereas context-sensitive premises like `c.LOCALS[x] = t` are deferred and collected in *prems*.

WrapBlock in line 5 takes the list of instantiated instructions and the unresolved premises *prems*. If any premise involves a control-flow label like `c.LABELS[l] = t*`, which arises from the Wasm instruction `br`, the instructions must be wrapped in a sufficient number of `block` constructs. The number of required nested blocks is determined by the highest label depth referenced in the premises. This function makes the instruction sequence include the necessary nesting structure potentially inserting dummy instructions to ensure the stack compatibility at block boundaries. Additionally, the added

`block` constructs may introduce additional premises on the module's types. If a block refers to a type index, for instance, it introduces a new premise like `c.TYPES[i] = blocktype`, which are resolved in the final module construction step.

Then, WRAPFUNC constructs a function that includes the generated block. It resolves the premises involving the types of the function and local variables, such as `c.RETURN = t*` and `c.LOCALS[x] = t`, which are introduced by instructions such as `return` and `local.get`, respectively. This step may introduce additional module-level conditions.

Finally, WRAPMODULE constructs a module that includes the generated function and components necessary to satisfy the remaining premises. For example, if the function contains `global.set x` and the premise `c.GLOBALS[x] = i32`, it generates a mutable global of type `i32` at the corresponding index. Likewise, premises like `c.FUNCS[i] = t` generate a dummy function at index `i` with the required type.

The framework is designed to be extensible to adapt to language changes, as it collects and resolves context-sensitive conditions in a modular and flexible manner. New forms of conditions like *tags* from the recent exception handling proposal [18] can be incorporated simply by extending the set of rules and default generators, without changing the core logic of the bottom-up strategy. In addition, while we currently use random sampling of instruction names (line 15, Algorithm 1), since the framework is capable of taking arbitrary instruction sequences as input, WEST can generate tests for specific instruction combinations or proposals. For example, one can provide a sequence of instructions from various proposals like `v128.add` [15], `return_call` [17], and `throw_ref` [18], then WEST can synthesize a minimal module embedding all of them and execute it in a valid runtime context. This capability is especially useful for edge-case exploration, regression reproduction, and proposal validation.

VI. EVALUATION

This section summarizes the evaluation result of WEST. We evaluate WEST with the following research questions:

- **RQ1 (Effectiveness)**: Can the generated tests reveal previously unknown bugs in real-world Wasm engines?
- **RQ2 (Validity)**: How often are the generated tests valid according to the target specification?
- **RQ3 (Diversity)**: How diverse are the generated tests in terms of specification coverage?

A. Experimental Setup

All experiments were done on servers running Ubuntu 20.04 with a 4.0 GHz Intel® Core™ i7-6700K processor and 32 GB of DDR4 RAM (Samsung 2133 MHz, 8 GB×4). We performed experiments on five major Wasm engines: V8 [27], SpiderMonkey [28], JavaScriptCore [29], Wasmtime [30], and Wasmer [31]. For all engines except Wasmer, we used the latest available commit as of May 2, 2025. Because Wasmer's latest commit was incompatible with Ubuntu 20.04, we used the most recent installable commit, released on Oct. 5, 2024.

TABLE I: Bugs discovered by generated tests.

Engine	Description	Status
Spider-Monkey	Importing a table with a non-nullable reference type is invalid	Fixed [33]
	`catch_ref` with non-null `exnref` is invalid	Fixed [34]
Java-Script-Core	GC instruction after parametric instruction causes parsing error	Fixed [35]
	`ref.null` after parametric instruction causes parsing error	Fixed [36]
	`try_table` after parametric instruction causes parsing error	Fixed [37]
	Accessing `v128` from null struct/array causes runtime crash†	Fixed [38]
	`throw_ref` with non-nullable reference is invalid	Fixed [39]
	Large `i31ref` with `v128` parameter gives incorrect result	Fixed [40]
	`v128` with block and `throw_ref` causes runtime crash	Dup. [41]
	Importing table with non-nullable type causes parsing error	Fixed [42]
	Importing `exnref` global causes runtime error	Report [43]
Wasm-time	`v128` array as table initializer causes runtime crash*	Dup. [44]
	Out-of-bounds access on `none` reference table does not trap*	Fixed [45]
	`global`/`table.set` with large `i31ref` causes runtime crash*	Dup. [46]
	Exporting function with long result type causes runtime crash	Fixed [47]
Wasmer	An empty data section causes parsing error	Dup. [48]

For each engine, we constructed a customized specification that includes only the features supported by that engine. V8 and SpiderMonkey support the Wasm 3.0 draft [20]. JavaScriptCore supports Wasm 3.0 except for multi-memory and memory64. Similarly, Wasmtime supports Wasm 3.0 except for exception handling. In contrast, Wasmer supports only Wasm 2.0. Each engine-specific specification was derived by selectively removing unsupported features from Wasm 3.0.

The WEST implementation [32] is about 6,000 additional lines of OCaml code on top of SpecTec. For each engine and the Wasm specification it supports, we generated tests for 24 hours with 3-second timeout for generating and executing tests. Tests were executed differently depending on target environments. For standalone Wasm runtimes (Wasmtime and Wasmer), tests were run directly. For browser-based engines (V8, SpiderMonkey, and JavaScriptCore), which do not natively support the `.wast` format, we translated each test script into an equivalent JavaScript harness using the official Wasm reference interpreter and executed the harness on the browser.

B. RQ1: Effectiveness

To evaluate the practical effectiveness of WEST, we executed the generated tests on their corresponding Wasm engines and investigated failing tests. A test was considered a failure if it caused a runtime crash, internal panic, validation mismatch, or different behavior than the expected result. We inspected the failing tests and automatically grouped them by their error messages and syntactic program patterns. Each bug produces distinctive and easily identifiable error message, and the programs that trigger these bugs also contain the distinctive features. Thus, bugs can be automatically classified using simple string manipulation.

As Table I shows, we found and reported 16 unique bugs from four of five target engines: two from SpiderMonkey, nine from JavaScriptCore, four from Wasmtime, and one from Wasmer. Three bugs from Wasmtime, annotated with ∗, were found in an older commit on Feb 25, 2025, and fixed in the

TABLE II: Validation success rates of generated tests.

Engine	#Generated	#Valid	Valid Rate
V8	352,001	320,884	91.16%
SpiderMonkey	52,810	48,065	91.01%
JavaScriptCore	429,192	393,983	91.80%
Wasmtime	413,932	376,306	90.91%
Wasmer	1,224,856	1,101,775	89.95%

TABLE III: AL coverage of official and generated tests.

Engine	Total	Official	Generated	Ratio
V8	2,876	2,097	2,071	98.76%
SpiderMonkey	2,876	2,097	2,059	98.19%
JavaScriptCore	2,876	2,097	2,085	99.43%
Wasmtime	2,780	2,015	1,993	98.91%
Wasmer	1,902	1,390	1,389	99.93%

latest commit. Most bugs are related to recently introduced features such as reference types, exception handling, and garbage collection (GC). Out of 16 bugs, four bugs were found to be duplicates of previously known bugs. Of the remaining 12 newly discovered bugs, 11 have been confirmed and fixed by developers, while one remains under review.

The fourth bug from JavaScriptCore, annotated with †, illustrates a subtle bug in the interaction between two recent Wasm proposals: GC and SIMD. For example, the following test attempts to access an element at index 0 from a `v128` array reference that is explicitly set to `null`:

```
(module
  (type $arr_v128 (array v128))
  (func $f (export "f") (result v128)
    (ref.null $arr_v128)
    (i32.const 0)
    (array.get $arr_v128))
)
(assert_trap (invoke "f") "")
```

The specification states that executing the test should result in a trap, which can be captured by `assert_trap`. However, executing it on JavaScriptCore results in a crash. This bug is triggered only when (1) the `array` is used (from GC proposal), (2) the element type is `v128` (from SIMD proposal), and (3) the access to a `null` reference is used. This illustrates how WEST can discover bugs from subtle interactions between different Wasm proposals in edge-case scenarios that are easy to overlook in manually written tests.

In addition to the bugs in Wasm engines, we also found two bugs in the official reference interpreter's validation logic, one bug in the `wast2js` converter, and two bugs in the specification itself. They are now fixed after our reports.

C. RQ2: Validity

To assess the ability to generate valid tests, we measured the proportion of valid tests generated for engine-specific specifications. We used the official Wasm reference interpreter as an oracle. Table II shows that around 90% of the tests generated by WEST are valid across all engines. Note that different engines generated different numbers of tests because of the differences in the input specifications and engine performance.

Manual inspection revealed that most validation failures stemmed from hard-coded heuristics (as described in Section IV-B). For example, WEST sometimes fails to filter out invalid modules due to over-approximation in context construction or insufficient handling for complex reference types. These issues could be resolved by improving our heuristics, but because such an improvement is not fundamental to the design of our system and has little impact on its effectiveness, we defer them as future work.

The results show that WEST mostly produces valid tests for a given specification, with invalid cases arising from a small number of fixable implementation details.

D. RQ3: Diversity

To evaluate the diversity and breadth of the generated tests, we measured their coverage over the engine-specific specifications. We used SpecTec's indirect interpreter to measure the instruction-level coverage of the specifications. Since the interpreter directly executes the semantics defined in an input specification, it can record which parts of the SpecTec-generated AL instructions are exercised by each test, which allows us to measure how thoroughly a test suite exercises the semantics of Wasm in a principled and fine-grained manner. We compared the coverage achieved by the generated tests to that of the official Wasm test suite. For each engine, we evaluated a subset of the official test suite corresponding to the features supported by that engine. Table III summarizes the results.

The **Total** column shows the total numbers of instruction-level AL nodes in the input specifications for the target engines. Note that the numbers for Wasmtime and Wasmer are smaller than the others, because they support smaller subsets of Wasm 3.0. Although JavaScriptCore does not support features such as multi-memory or memory64, its total instruction count remains identical to V8 and SpiderMonkey due to the parametric nature of the specification written in SpecTec, where parameterized rules (e.g., over memory indices or address widths) cover a wide range of behaviors using shared rules.

The **Official** and **Generated** columns show the numbers of AL instructions covered by the official and generated tests, respectively. Note that even the execution of official tests does not fully cover the specifications, as some rules, such as auxiliary functions that are only used in validation rules, are not reachable during execution.

Finally, the **Ratio** column shows the relative coverage of the generated tests compared to the official suite. The ratio exceeds 98% for all engines, demonstrating that the coverage of the automatically generated tests is comparable to that of the official hand-written test suite across all engines.

E. Threats to Validity

1) Bug Classification Accuracy: Bug classification is based on automated grouping of error messages. This may occasionally result in misclassification of similar failures. Furthermore, some instructions (e.g., involving `NaN` or `relaxed_SIMD`)

are nondeterministic by nature. Since WEST uses one possible behavior observed from the SpecTec interpreter to generate assertions, valid executions may still lead to test failures. Therefore, we applied similar heuristic filters to reduce false positives regarding nondeterminism, which may also exclude genuine bugs. These factors do not compromise the reported bug-detection capability of WEST, but may underreport the total numbers of discoverable bugs in practice.

2) Correctness of Reported Bugs: Our automated bug classification may overcount bugs by treating tests with different symptoms but with the same root cause as distinct issues. As we discussed in Section VI-B, we have reported all findings to the respective maintainers. Out of 18 unique bugs found, 13 have been confirmed and five are pending for confirmation.

3) Specification Coverage as a Metric: We use the runtime coverage of a specification as a proxy for test diversity. As briefly mentioned in Section VI-D, due to the parametric nature of Wasm specifications written in SpecTec, the specification coverage may not accurately represent the actual dynamic coverage. Thus, the specification coverage is an approximation of test diversity, and a more fine-grained assessment would require deeper engine-specific coverage metrics.

F. Discussions

There are several limitations of WEST regarding the automation and the generality. While WEST automates test generation from mechanized specifications, it is not entirely specification-agnostic. The current implementation relies on several Wasm-specific heuristics (Section IV-B), and thus may require further engineering effort to generalize to future extensions of the Wasm specification. Also, WEST relies on SpecTec for handling Wasm specifications. Thus, the system supports only specifications expressed in the SpecTec DSL. Significant changes to the structures of validation rules, execution semantics, or SpecTec's internal representation of specifications may require additional adjustments to the test generation framework.

The primary goal of this paper is not to demonstrate superiority over existing fuzzers in terms of generation throughput or validation rates, but to showcase a new generation strategy based on mechanized specifications. Unlike existing fuzzers, WEST focuses on specification-adaptive generation across varying spec versions and engine configurations. This use case is orthogonal to existing fuzzers, and we believe it brings complementary strengths. Therefore, head-to-head comparisons with other tools are not the focus in this paper and deferred to the future work.

While instruction-level coverage between WEST and the official test suite is similar, WEST brings several advantages beyond the coverage: 1) it supports automatic test generation as the spec evolves, 2) produces test cases tailored to specific engines or Wasm subsets, and 3) allows guided construction (e.g., with instruction targets for bottom-up generation), enabling focused testing of rarely exercised semantics or their combinations. These properties make WEST a natural complement to the official suite, especially for conformance testing under evolving or restricted specification variants.

Beyond serving as a test generator, WEST also functions as a validation tool for specifications themselves, identifying inconsistencies with implementations through generated tests. We believe that this specification-based approach offers a principled path toward more robust, reusable, and future-proof infrastructure for testing language implementations.

As future work, we plan to extend WEST beyond post-hoc testing of finalized implementations. By integrating our framework more tightly into the development process, we aim to support interactive test generation and validation during specification authoring and early-stage implementation. Additionally, we plan to explore coverage-guided fuzzing techniques, support for more expressive semantic interaction scenarios, and broader application to other specification-driven domains beyond Wasm.

VII. Related Work

This section discusses related work on Wasm test generation, specification-guided testing, and mechanized specifications. We compare WEST to existing fuzzing tools and general frameworks, highlighting the key differences.

A. WebAssembly Test Generation and Fuzzing

Researchers have developed automatic testing techniques for Wasm engines. One of the most widely used tools is wasm-smith [9], which generates syntactically valid and structurally diverse Wasm modules. Mutation-based fuzzers such as wasm-mutate [10], WASMaker [11], and Wapplique [12] use real-world Wasm programs as seeds and apply various mutation strategies to produce variant programs. While existing tools rely on manually implemented rules, which must be updated whenever specifications change, WEST derives all structural and semantic constraints directly from specifications, enabling *specification-adaptive* test generation. Moreover, while traditional fuzzers often depend on differential testing to detect inconsistencies, WEST interprets specifications' runtime semantics directly and can compute expected execution results.

B. Specification-Guided Testing

Our work also relates to broader efforts in leveraging formal specifications to guide automated testing. Tools such as Csmith [49], LangFuzz [50], and Grammar-based Whitebox Fuzzing [51] have explored the use of grammars and semantic constraints together to generate test inputs for complex languages like C and JavaScript. While these tools rely on manually implemented models, WEST operates directly on a formal, executable specification, enabling greater generality and long-term maintainability.

WADIFF [13] is the most closely related work to ours in terms of both Wasm testing and spec-based testing. It introduces a custom DSL to represent the prose Wasm specification and applies symbolic execution and mutation strategies to generate tests. While conceptually similar, WADIFF is based on a manually re-written specification of Wasm, which makes

it difficult to capture the full semantics of complex instructions such as `br` and `if`. In contrast, WEST directly consumes the official, mechanized specification written in SpecTec, allowing it to fully support all Wasm instructions, including proposals introduced after version 2.0.

C. Mechanized Specifications

Our work also relates to broader efforts in mechanizing programming language semantics. Frameworks such as K [52] [53], PLT Redex [54], Rocq(Coq) [55], [56], and Isabelle/HOL [57] have been used to formalize a wide range of language semantics including Wasm, and to derive tools such as interpreters, analyzers, and test generators. One key distinctive feature of WEST is that it operates directly on Wasm's official mechanized specification, not a re-encoded approximation.

In the context of JavaScript, ESMeta [58], [59] and JEST [60] offer mechanized specifications and test generation frameworks, respectively, built from the official JavaScript language specification. SpecTec plays a similar role for Wasm, providing a structured, machine-readable representation of the language's syntax, validation rules, and runtime semantics. By building on SpecTec, WEST inherits the benefits of mechanized specifications without requiring manual reimplementation of validation or execution logic. This tight coupling between specification and testing infrastructure distinguishes our approach from prior work and provides a robust foundation for supporting future language extension.

VIII. Conclusion

As the Wasm ecosystem continues to evolve and incorporate increasingly complex features, ensuring the correctness of Wasm engines has become more challenging than ever. However, manual test development remains labor-intensive and difficult to scale. In this work, we presented WEST, a specification-based test generation framework that systematically generates test programs conforming to the language grammar and validation rules, directly from mechanized specifications written in SpecTec. Our approach requires minimal manual adaptation to target different versions or subsets of the Wasm specification, making it well-suited for testing a diverse range of engines and evolving language features. Our experimental evaluation demonstrated that WEST can uncover real-world, previously unknown bugs across five production-level Wasm engines, achieve high validity rates for generated tests, and provide specification-level coverage comparable to the official hand-written test suite. These results demonstrate the practicality and effectiveness of using mechanized specifications as the basis for systematic and scalable test generation.

Acknowledgment

This work was partly supported by the National Research Foundation of Korea (NRF) (2022R1A2C2003660 and 2021R1A5A1021944), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (2024-00337703), and Samsung Electronics Co., Ltd.

References

[1] 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 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363

[2] A. Rossberg and WebAssembly Community Group. (2025) Webassembly specification (release 2.0). [Online]. Available: https://webassembly.github.io/spec/core/

[3] K. Varda. (2017) Introducing cloudflare workers: Run javascript service workers at the edge. [Online]. Available: https://blog.cloudflare.com/introducing-cloudflare-workers/

[4] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[5] A. Tekeli. (2022) Webassembly (wasm) in blockchain. [Online]. Available: https://blog.devgenius.io/webassembly-wasm-in-blockchain-f651a8ac767b

[6] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of webassembly for embedded systems," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4.

[7] S. Beeker. (2025) Server side webassembly exploring the unknown. [Online]. Available: https://dev.to/pullreview/server-side-web-assembly-exploring-the-unknown-26la

[8] WebAssembly Community Group. (2025) WebAssembly Core Testsuite. [Online]. Available: https://github.com/WebAssembly/spec/tree/main/test/core

[9] B. Alliance. (2023) wasm-smith: A webassembly test case generator. [Online]. Available: https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith

[10] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-mutate: Fast and effective binary diversification for webassembly," *Computers & Security*, vol. 139, p. 103731, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824000324

[11] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1262–1273. [Online]. Available: https://doi.org/10.1145/3650212.3680358

[12] W. Zhao, R. Zeng, and Y. Zhou, "Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1035–1047. [Online]. Available: https://doi.org/10.1145/3650212.3680340

[13] 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*, ser. ASE '23. IEEE Press, 2024, p. 939–950. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00188

[14] J. Park, Y. Kim, and I. Yun, " RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 920–938. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00003

[15] D. Gandluri and WebAssembly Community Group. (2021) Simd proposal for webassembly. [Online]. Available: https://github.com/WebAssembly/simd/

[16] B. Smith, C. Watt, and WebAssembly Community Group. (2023) Threads Proposal for WebAssembly. [Online]. Available: https://github.com/WebAssembly/threads/

[17] A. Rossberg and WebAssembly Community Group. (2023) Tail call proposal for webassembly. [Online]. Available: https://github.com/WebAssembly/tail-call/

[18] H. Ahn and WebAssembly Community Group. (2023) Exception handling proposal for webassembly. [Online]. Available: https://github.com/WebAssembly/exception-handling/

[19] A. Rossberg and WebAssembly Community Group. (2023) Gc proposal for webassembly. [Online]. Available: https://github.com/WebAssembly/gc/

[20] ——. (2025) WebAssembly Specification Release 3.0 (Draft 2025-05-15). [Online]. Available: https://webassembly.github.io/spec/versions/core/WebAssembly-3.0-draft.pdf

[21] WebAssembly Community Group. (2025) Webassembly proposals. [Online]. Available: https://github.com/WebAssembly/proposals

[22] D. Youn, W. Shin, J. Lee, S. Ryu, J. Breitner, P. Gardner, S. Lindley, M. Pretnar, X. Rao, C. Watt, and A. Rossberg, "Bringing the webassembly standard up to speed with spectec," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656440

[23] SpecTec Team. (2025) Wasm SpecTec. [Online]. Available: https://github.com/Wasm-DSL/spectec/tree/main/spectec

[24] A. Rossberg. (2025) SpecTec has been adopted. [Online]. Available: https://webassembly.org/news/2025-03-27-spectec/

[25] A. Rossberg and WebAssembly Community Group. (2022) Profiles Proposal for WebAssembly. [Online]. Available: https://github.com/WebAssembly/profiles/

[26] S. Ceri, G. Gottlob, and L. Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Trans. on Knowl. and Data Eng.*, vol. 1, no. 1, p. 146–166, Mar. 1989. [Online]. Available: https://doi.org/10.1109/69.43410

[27] Google V8 Team. (2025) V8 JavaScript Engine. Commit 4c7d435, accessed May 2, 2025. [Online]. Available: https://github.com/v8/v8/commit/4c7d435

[28] Mozilla SpiderMonkey Team. (2025) SpiderMonkey JavaScript Engine. Commit 22ae5ab, accessed May 2, 2025. [Online]. Available: https://github.com/mozilla/gecko-dev/commit/22ae5ab

[29] Apple WebKit Team. (2025) JavaScriptCore (JSC). Commit 9d02067, accessed May 2, 2025. [Online]. Available: https://github.com/WebKit/WebKit/commit/9d02067

[30] Bytecode Alliance. (2025) Wasmtime. Commit 2c2e7cf, accessed May 2, 2025. [Online]. Available: https://github.com/bytecodealliance/wasmtime/commit/2c2e7cf

[31] Wasmer Team. (2024) Wasmer. Version released on October 5, 2024. [Online]. Available: https://github.com/wasmerio/wasmer/releases/tag/v4.4.0

[32] WEST Team. (2025) WEST Artifact Repository. [Online]. Available: https://github.com/ase25-west/west

[33] seyoon1705. (2025) Wasm validation failure when importing table with non-nullable reference type. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1966552

[34] ——. (2025) validation failure: catch_ref passes a non-nullable exnref, not a nullable exnref. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1967661

[35] ——. (2025) Javascriptcore raises 'invalid extended gc op' error for valid wasm module. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293204

[36] ——. (2025) Wasm module compile error when a function contains 'ref.null' after 'return'. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293107

[37] ——. (2025) Wasm module compile error when a function contains 'try_table' after 'return'. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293106

[38] ——. (2025) Getting v128 field from null struct/array results in runtime crash. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293205

[39] ——. (2025) Validation failure when a function contains 'ref.null exn', 'ref.as_non_null', and 'throw_ref'. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293211

[40] ——. (2025) [wasm] wrong reference equality check result. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293207

[41] ——. (2025) Using v128 as local/param, block instructions, and throw_ref results in runtime crash. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293571

[42] ——. (2025) Wasm validation failure when importing a table with a non-nullable reference type. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293030

[43] ——. (2025) Cannot get value of exnref global. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=293340

[44] fitzgen. (2025) Wasm gc: Unify supported alignment and minimum block size in freelist. [Online]. Available: https://github.com/bytecodealliance/wasmtime/pull/10349

[45] f52985. (2025) Out of bounds table access for none reference does not trap. [Online]. Available: https://github.com/bytecodealliance/wasmtime/issues/10353

[46] fitzgen. (2025) Wasm gc: Fix is-null-or-i31ref checks. [Online]. Available: https://github.com/bytecodealliance/wasmtime/pull/10221

[47] f52985. (2025) Codegen fails when exporting a function with long result type. [Online]. Available: https://github.com/bytecodealliance/wasmtime/issues/10741

[48] W. Team. (2025) Bugs of wasmer found by bottom-up west. [Online]. Available: https://github.com/ase25-west/west/blob/main/spectec/bugs/bottomup/wasmer/summary.md

[49] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[50] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 38. [Online]. Available: https://dl.acm.org/doi/10.5555/2362793.2362831

[51] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, p. 206–215, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375607

[52] R. Verification. (2013) K semantic framework. [Online]. Available: https://kframework.org/

[53] R. V. Inc., "KWasm: A Formal Semantics of WebAssembly in K," https://github.com/runtimeverification/wasm-semantics, 2025, accessed: 2025-05-29.

[54] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[55] R. Krebbers, "Aliasing Restrictions of C11 Formalized in Coq," in *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2013, p. 50–65. [Online]. Available: https://doi.org/10.1007/978-3-319-03545-1_4

[56] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner, "Two mechanisations of webassembly 1.0," in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 61–79. [Online]. Available: https://doi.org/10.1007/978-3-030-90870-6_4

[57] C. Watt, M. Trela, P. Lammich, and F. Märkl, "Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591224

[58] J. Park, J. Park, S. An, and S. Ryu, "JISET: JavaScript IR-based semantics extraction toolchain," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 647–658. [Online]. Available: https://doi.org/10.1145/3324884.3416632

[59] (2022) ESMeta: An ECMAScript specification metalanguage used for automatically generating language-based tools. [Online]. Available: https://github.com/es-meta/esmeta

[60] 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 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE. New York, NY, USA: Association for Computing Machinery, 2021, pp. 13–24. [Online]. Available: https://ieeexplore.ieee.org/document/9402086