LWDIFF: An LLM-Assisted Differential Testing Framework for WebAssembly Runtimes

Shiyao Zhou[1], Jincheng Wang[1], He Ye[2], Hao Zhou[1], Claire Le Goues[3], Xiapu Luo[1]
[1] The Hong Kong Polytechnic University, Hong Kong, China
[2] University College London, London, United Kingdom
[3] Carnegie Mellon University, Pittsburgh, United States
{csszhou1, csxluo}@comp.polyu.edu.hk, wjcuhk@gmail.com, he.ye@ucl.ac.uk, hcnzhou@polyu.edu.hk, clegoues@cs.cmu.edu

***Abstract*—WebAssembly (Wasm) runtimes execute Wasm programs, a popular low-level language for efficiently executing high-level languages in browsers, with broad applications across diverse domains. The correctness of those runtimes is critical for both functionality and security of Wasm execution, motivating testing approaches that target Wasm runtimes specifically. However, existing Wasm testing frameworks fail to generate test cases that effectively test all three phases of runtime, i.e., decoding, validation, and execution. To address this research gap, we propose a new differential testing framework for Wasm runtimes, which leverages knowledge from the Wasm language specification that prior techniques overlooked, enhancing comprehensive testing of runtime functionality. Specifically, we first use a large language model to extract that knowledge from the specification. We use that knowledge in the context of multiple novel mutation operators that generate test cases with diverse features to test all three runtime phases. We evaluate LWDIFF by applying it to eight Wasm runtimes. Compared with the state-of-the-art Wasm testers, LWDIFF achieves the highest branch coverage and identifies the largest number of bugs. In total, LWDIFF discovers 31 bugs across eight runtimes, all of which are confirmed, with 25 of them previously undiscovered.**

I. INTRODUCTION

WebAssembly (Wasm) is a low-level, stack-based, assembly-like language that allows high-level languages like C, C++, and Rust to be executed in the browser at near-native performance. The World Wide Web Consortium (W3C) has recommended Wasm as an official web standard, the fourth language to run natively in browsers (after HTML, CSS, and JavaScript). Beyond its use for web applications, Wasm is increasingly gaining adoption across other domains, e.g., in blockchain technology [1], [2], [3].

While the Wasm specification defines a portable binary language and a corresponding text format, a Wasm *runtime* is the key to decoding, validating, and executing Wasm binaries. Many implementations of the Wasm runtime exist, catering to a variety of contexts and use cases, such as performance optimization or security [4], [5], [6], [7], [8]. Defects in Wasm runtime implementations can be critical, leading to incorrect calculation results or runtime crashes [9], motivating the development of automatic defect detection [10], [11].

Differential testing offers an attractive potential approach for identifying defects in Wasm runtime implementations. Differential testing compares multiple implementations of the same specification to find behavioral differences (e.g., bugs, inconsistencies, or performance differences) [12], [13], [14], [15]. In our context, differential testing involves executing the same Wasm program across multiple runtimes simultaneously, and comparing their outputs. If any runtime produces a different result or behavior compared to others, it indicates a potential bug [11], [16].

Executing a Wasm program involves three runtime phases: decoding, validation, and execution. Comprehensive testing of all three phases requires test cases with diverse features, including those that: (1) violate encoding rules (to test decoding); (2) violate validation rules (to test validation), and (3) those that both trigger many instruction execution violating the encoding rules, to test the decoding phase; (3.a) trigger as many instruction execution behaviors as possible, and (3.b) contain sequences of instructions with diverse control flows (to test execution). Although promising, the many previously-proposed approaches for both differential and fuzz testing of Wasm runtimes [17], [11], [16], [18], [12] do not produce all of these necessary test types. For example, WADIFF [11] generates test cases for each individual Wasm instruction without control flow constructs, but fails to generate tests that contain combinations of instructions, or diverse control flow.

In this paper, we propose LWDIFF, a differential testing framework that finds bugs in Wasm runtimes. LWDIFF entails three key insights to overcome central challenges to effectively testing Wasm runtimes:

a) Automatic extraction of key information from the language specification: Generating diverse test cases benefits from knowledge extracted from the language specification. For example, to exercise whether an instruction is implemented correctly, we need knowledge about how the instruction is validated and executed to guide the test case generation. Specifically, we need to use this knowledge to create representative test cases, including those with invalid instructions, for testing validation, and those with valid instructions, for testing execution by triggering as many execution behaviors as possible. However, because the Wasm specification is in natural language, extracting the necessary knowledge and using it to guide the test case generation is challenging [11]. Previous approaches require manually designed patterns to extract knowledge [11] or require the developers to encode the knowledge directly [17], [16]. Both approaches are laborious and error-prone.

Instead, inspired by recent advancements in large language model (LLM), we propose to use an LLM to analyze the Wasm specification [19], extract necessary knowledge, and inform the construction of multiple mutation operators to generate diverse test cases to test all three phases of runtime.

b) A position-aware instruction insertion mutation strategy: Testing each instruction's potential runtime outcomes comprehensively is challenging. In addition to implementing the behaviors required by the specification, the runtime may implement customized behaviors (e.g., optimizations for control flow) for instructions at specific positions. Techniques that overlook the impact of position on behavior [11] fail to test these customized behaviors. To test the implementation of each instruction comprehensively, we design a position-aware insertion strategy to generate test cases where an instruction is placed at different potentially vulnerable positions.

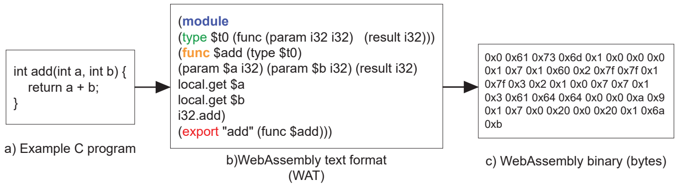
c) Modeling mutation scheduling as a multi-armed (MAB) problem: Generating test cases with different features commonly requires multiple mutation operators [20], [21], [22], [23]. However, scheduling different mutation operators is non-trivial, because they can have varying effectiveness [20], [24]. A common method is to select mutation operators randomly [22], [25], [16]. However, this approach cannot evaluate the mutation operators and select the optimal one, leading to inefficient testing. Instead, to identify and select the optimal mutation operator, we model mutation scheduling as a multi-armed bandit (MAB) problem [26]. We design a coverage-based reward function to evaluate each mutation operator's historical performance and propose an MAB-based selector to select the optimal mutation operator.

We implement LWDiff and apply it on eight Wasm runtimes from four well-known Wasm runtime projects, including Wasmer [27], Wasmtime [4], WasmEdge [6], and five modes of WAMR [28]. As a result, in 24 hours, LWDiff generates 262,531 test cases triggering differences among runtimes. By analyzing the differences, LWDiff succeeds in identifying 31 bugs, and all of them are confirmed and fixed. Compared with the state-of-the-art baselines, LWDiff achieves the highest code coverage and identifies the largest number of bugs, illustrating the effectiveness of LWDiff.

The main contributions of this paper are as follows.

- **Effective Mutation Operators**. We leverage an LLM to extract various types of knowledge from a language specification. We use that knowledge to design multiple mutation operators capable of generating test cases with diverse features. These test cases cover all three phases of executing a Wasm program: decoding, validation, and execution, contributing to effective testing.
- **LWDiff**. We propose a differential testing framework for Wasm runtimes, and implement it in a prototype, LWDiff. LWDiff addresses several technical challenges. We propose a position-aware insertion strategy to test the implementation of each instruction comprehensively. Additionally, we design an MAB-based mutation operator selector to identify and select the optimal mutation, ensuring efficient testing. The implementation of our approach is publicly available at https://github.com/erxiaozhou/LWDIFF2024.
- **Comprehensive Evaluation**. We comprehensively evaluate LWDiff on eight Wasm runtimes to demonstrate its effectiveness, and LWDiff outperforms the state-of-the-art tools. With the help of LWDiff, we have discovered 31 bugs. 25 of them are unseen before, and all the bugs are confirmed and fixed by the developers.

II. WebAssembly



```
int add(int a, int b) {
    return a + b;
}
```

a) Example C program

```
(module
(type $t0 (func (param i32 i32)   (result i32)))
(func $add (type $t0)
(param $a i32) (param $b i32) (result i32)
local.get $a
local.get $b
i32.add)
(export "add" (func $add)))
```

b)WebAssembly text format (WAT)

```
0x0 0x61 0x73 0x6d 0x1 0x0 0x0 0x0
0x1 0x7 0x1 0x60 0x2 0x7f 0x7f 0x1
0x7f 0x3 0x2 0x1 0x0 0x7 0x7 0x1
0x3 0x61 0x64 0x64 0x0 0x0 0xa 0x9
0x1 0x7 0x0 0x20 0x0 0x20 0x1 0x6a
0xb
```

c) WebAssembly binary (bytes)

Fig. 1. Wasm example; a simple C function (left) compiled into the Wasm (middle and right). `local.get` and `i32.add` are the opcode of the instructions, and `$a` and `$b` are the immediate arguments.

WebAssembly (Wasm) is a portable, low-level assembly-like binary language definition and corresponding text format for executable programs [19]. It is largely intended to support portable, high-performance web applications. Therefore, it is seeing application in a diversity of domains [29], [30], [31], [32]. In this section, we provide background on key concepts in the Wasm language specification and in Wasm runtimes, which are necessary for understanding our contribution.

A. The Wasm Language

Fig. 1 shows an example, illustrating how a simple `add` function written in C code is compiled to Wasm. Wasm code can be represented in the Wasm Text Format (WAT, middle, a human-readable text format) or the Wasm byte code (right). Wasm bytecode fundamentally comprises units of computation consisting of instruction sequences to be executed by the given Wasm runtime. Compiled Wasm code is organized into a module, a unit of deployment and execution [19], consisting of up to 13 sections (including sections for, e.g., code, type, global, export) in a specified order.

Our small C example's Wasm module contains four sections, including a `type` section defining the types used in the program, a `function` section declaring the type of each function, a `code` section defining each function, and an `export` section to export the definitions (e.g., functions) externally. In this example, the `type` section defines a type `$t0` that takes two `i32` (32-bit integer) parameters and returns an `i32`; the `function` section uses the type `$t0` defined in the type section to declare the `add` function. A Wasm function consists of a type, local variables, and an instruction sequence. Not all sections are required for all programs; for example, since this program does not use dynamically managed memory, there is no `memory` section.

In general, a Wasm instruction consists of an opcode and immediate arguments [19]. Wasm is stack-based: each instruction implements its functionality by taking operands from the stack, and pushing any result back onto the stack [19]. For example, the `i32.add` instruction takes two operands from the stack, adds them, and pushes the result back onto the stack.

B. *Wasm Runtime*

A Wasm runtime is the key component in the ecosystem, and is responsible for decoding, validating, and executing Wasm programs [19]. First, the runtime *decodes* Wasm bytecode into a module. Then, the runtime *validates* the module, checking whether the code in the module conforms to validation rules specified in the Wasm language specification. Finally, if the module is valid, the runtime *executes* it. The runtime creates a module instance according to the definitions in the module and then executes the specified exported function (hereafter referred to as the *entry* function). During the execution of this function, the runtime processes the instruction sequence according to its control flow.

At a high level, then, to execute a Wasm program correctly, a runtime must (1) validate each instruction as described in the specification, and (2) properly manage the Wasm binary's control flow and execute each instruction correctly. To illustrate, Fig. 2 shows example validation rules for `i32.store`: If linear memory is defined in the context (line 1), the immediate argument `alignment` is a power of 2 and not greater than 4 (line 2), and there are two operands of type `i32` on the stack (line 3), the instruction `i32.store` is valid. The specification also introduces how the instruction input (i.e., immediate arguments and operands) determines its execution behavior [19], [11]. For example, as shown in Fig. 3, for the instruction `i32.store`, the immediate argument `offset` and the second operand (denoted as i) determine the address for storing a byte (line 7), leading to two possible execution behaviors. If the address is out of memory bounds, the execution will terminate and lead to a trap, i.e., an exception (lines 8 and 8.a). Otherwise, the instruction executes successfully, storing a byte (line 10).

To date, a variety of Wasm runtimes are available and under active development, including Wasmtime [4], Wasmer [27], WAMR [28] and others, offering tradeoffs in terms of target environments (such as edge computing environments and blockchain systems) [33], [7], [34], [5].

III. LWDIFF

Fig. 4 provides an overview of LWDIFF, which works as follows. Before testing, LWDIFF first analyzes Wasm documentation to extract structured knowledge of the language specification to inform mutation (Section III-A). The mutation engine (Section III-B) uses this extracted knowledge to implement three types of mutation strategies, including mutation operators that are intended to generate test cases covering all three Wasm runtime phases (i.e., decoding, validation, and execution). Meanwhile, we maintain a *seed pool*, which stores generated *test seeds*, where a seed is a Wasm binary.

```
1. The memory C.mems[0] must be defined in the context.
2. The alignment $2^{memarg.align}$ must not be larger than 4.
3. Then the instruction is valid with type [i32 i32] ->
     [].
```

Fig. 2. Simplified validation rules of `i32.store`

```
1. Let F be the current frame.
2. Let a be the memory address F.module.memaddrs[0].
3. Let mem be the memory instance S.mems[a].
4. Pop the value i32.const c from the stack.
5. Pop the value i32.const i from the stack.
6. Let ea be the integer i + memarg.offset.
7. Let N be the bit width |i32| of number type i32.
8. If ea + N/8 is larger than the length of mem.data,
     then:
  8.a. Trap.
9. Let b* be the byte sequence $\mathrm{bytes}_{i32}(c)$.
10. Replace the bytes mem.data[ea:N/8] with $b*$.
```

Fig. 3. Simplified execution behavior of `i32.store`

In each round of the testing campaign, LWDIFF first randomly selects a seed from the seed pool and applies one of the mutation operators to that seed to generate a new test case. We model the selection problem as a multi-arm bandit game, using the UCB1-Tuned algorithm [35] to guide mutation selection (Section III-C). The differential testing engine (Section III-D) then executes the generated test case on different Wasm runtimes to identify a difference, which is a potential bug. Finally, the test case is evaluated: if it improves code coverage over the existing seeds, it is added to the seed pool for later testing rounds (Section III-E).

A. *Wasm Specification Analysis*

In this section, we describe how we use an LLM to extract language specification knowledge from Wasm documentation.

Test case generation for language runtimes or compilers typically benefits from an understanding of the underlying language specification [11], [14], [36], [37]. In the Wasm runtime testing context, for example, an understanding of instruction validation rules can guide mutation to insert instructions that either follow those rules, or deliberately violate them; doing so can enable explicit testing of the Wasm runtime validation phase. Prior work for Wasm testing has required either manual specification of rules to extract this kind of knowledge [11], or for programmers to encode the knowledge directly [17], [16]. While effective, such manual analysis is both labor-intensive and error-prone, and liable to inevitable incompleteness.

LWDIFF instead uses LLM with retrieval-augmented generation [38] to automatically analyze Wasm documentation (e.g., the official specification) to extract and summarize key language specification knowledge. Generally speaking, LLM augmentation methods rely on a knowledge retriever to segment provided documents (such as the Wasm specification) and conduct a semantic search, identifying knowledge chunks relevant to the downstream task [38]. These chunks, along with downstream task descriptions, are then provided as inputs to an LLM to derive a task solution (e.g., the validation rules relevant to a particular instruction).

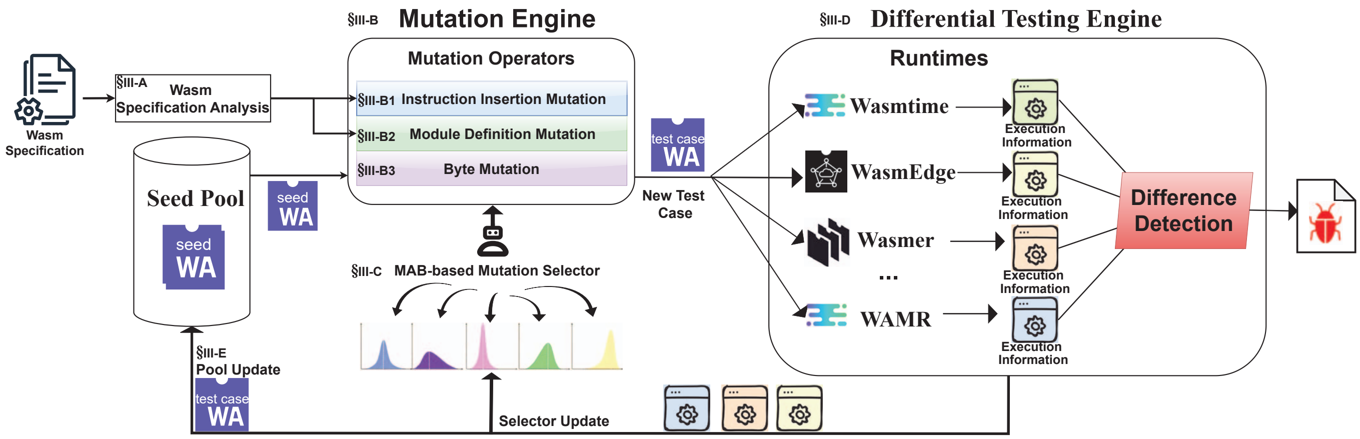


Fig. 4. An overview of LWDiff

Knowledge Retriever. The knowledge retriever, which is designed for retrieving relevant information from the Wasm specification, consists of four steps: 1) documentation segmentation; 2) query formulation; 3) semantic embedding and 4) knowledge retrieval. First, we segment the documentation into chunks. Typically, knowledge retrieval accuracy relies heavily on chunking configurations, such as the maximum tokens per chunk. This often necessitates frequent (and tedious) manual adjustments. We leverage the well-structured nature of the Wasm specification documentation, and segment it according to its hierarchical outline. We parse the documentation's source code (e.g., .rst files) using Docutils [39] to create a hierarchical outline. Each entry in the hierarchy corresponds to a subsection, capturing the title and the text as a chunk. Second, query formulation generates natural language queries, such as *"The validation of the instruction i32.add"*. Then, both the chunk titles (i.e., section names) and the query are converted into dense vector representations (embeddings). Finally, we calculate the cosine similarity between these embeddings to identify the best-matching subsection (based on title relevance) and retrieve its textual content as relevant knowledge.

Background-augmented Prompting. LWDiff uses background-augmented prompts to extract knowledge to inform mutation operator construction, without a need for fine-tuning. Background-augmented prompts produce six key types of knowledge necessary to construct mutation operators: 1) *instruction format*; 2) *instruction validation rules*; 3) *instruction type*; 4) *instruction execution behavior conditions* (i.e., the conditions under which an instruction's behavior is triggered); 5) *control flow constructs*; and 6) *module definition formats*. Prompts include task-specific descriptions (e.g., "summarize the type of the instruction"), background information (e.g., validation and execution of instruction), and expected response formats. The output in response to these prompts is structured knowledge for constructing mutation operators. Further detail on mutator construction given this knowledge is provided in Section III-B.

Format Refinement for LLM Responses. LLM responses do not always conform to an expected format, which can hinder its use in downstream tasks. To address this issue, we adopt an iterative approach to guide the LLM to correctly structure its responses. We design a rule-based format checker to identify ill-formatted responses, providing a description of the detected formatting issue. For example, the format checker verifies whether the LLM represents the validation rules for an instruction as a list of mappings with specified keys. If a response is identified as ill-formatted, we append the issue description to the prompt, and prompt the LLM to revise its response. This process is repeated until either the response adheres to the expected format or a maximum number (e.g., 3) of retries is reached. If the final response is incorrect, we manually correct it. We discuss extracted specification correctness in Section VI-B.

Overall, the LLM processes 182 PDF pages of Wasm documentation to extract knowledge regarding 424 instructions, 13 types of module definitions, and three control flow constructs.

B. Mutation Engine

Given a seed test case from the seed pool, the mutation engine begins by selecting a mutation operator to apply (Section III-C describes how). Each mutation, as applied to a test seed, produces a single new test case. In this section, we describe how we design mutation operators to generate test cases covering all three phases of a Wasm runtime. We first detail LWDiff's mutation strategies (Sections III-B1–III-B2) that use the LLM-extracted knowledge to create test cases for both the validation and execution phases. Then, we describe how we design the mutation operators to generate test cases to test the decoding phase (Section III-B3).

1) Instruction Insertion Mutation: The goal of the instruction insertion mutation strategy is to test whether a runtime can correctly validate and execute instructions. We design two insertion-based mutation operators for inserting a *Target Instruction* or *Code*. The *Target Instruction Insertion Mutation* helps test whether a runtime correctly validates and executes an instruction; the *Code Insertion Mutation* helps test whether the runtime can handle diverse control flow.

Target Instruction Insertion Mutation aims to generate test cases that comprehensively trigger behaviors of a *target instruction* (e.g., `i32.store` or `i32.add`) during validation and execution, thereby verifying its correct implementation.

To achieve this, there are two challenges to address. First, we aim to test the full suite of specified instruction behaviors described in the language specification. To address this, we use the extracted specification knowledge to identify a set of representative inputs (I_s) for the *target instruction*, which can lead to test cases triggering these behaviors. Second, it is also challenging to test runtime behaviors that the specification does not explicitly describe. This is because a runtime [28] may have customized behaviors for an instruction at a specific position. To mitigate this issue, we place the *target instruction* at potentially vulnerable positions by inserting the *target instruction* into different positions (P_s) and wrapping the *target instruction* in various methods (W_s).

(1) Overall Workflow. Before testing, given a *target instruction* with the opcode Op, we determine the mutations to apply by first identifying I_s, P_s, and W_s, combined to create a set of mutation parameters ($Paras$), represented as:

$$Paras = \{Para(Op, I, P, W) \mid I \in I_s, P \in P_s, W \in W_s\}$$

During testing, we generate a new test case according to a mutation parameter $Para(Op, I, P, W)$ as follows. First, we determine the *target instruction* by combining the opcode (Op) and the immediate arguments and determine the *setup instructions* needed to set the operands (all included in I). Then, we wrap the *target instruction* with the wrapping method (W). Finally, we insert the wrapped *target instruction* and the *setup instructions* into the position (P) to generate a test case. We prioritize mutation parameters that have not been previously selected; once all mutation parameters have been selected, we then choose one randomly.

(2) Specification-Driven Representative Input Extraction. Triggering a specific behavior requires specific inputs (i.e., immediate arguments and operands) [11]. For example, test `i32.div_s` comprehensively requires at least one case where the second operand is 0 (to test that the runtime correctly raises an exception). The specification provides information on how an instruction is validated and executed; the challenge lies in extracting and using this knowledge correctly [11].

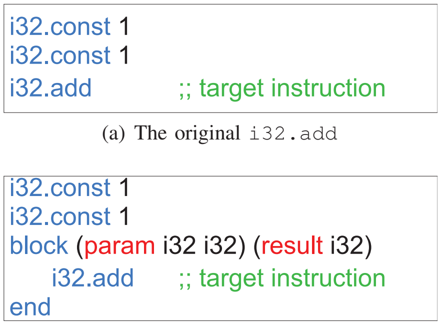
We address this use the LLM-extracted validation rules and instruction execution behaviors (recall the examples in Section II) to determine representative instruction inputs (I_s). First, we determine the instruction format, i.e., the number and types of operands and immediate arguments. For example, the instruction `i32.store` requires two immediate arguments of type `u32`, and two operands of type `i32`. Second, we determine representative inputs that lead to representative behaviors, covering both valid/invalid formulations, and execution behaviors. We ask the LLM to identify conditions that the instruction inputs must meet for an instruction to exhibit a specific behavior. Given the structured response from the LLM, we automatically build symbolic constraints over operands and immediate arguments to describe necessary conditions to trigger a given behavior. For example, the behavior of the instruction `i32.store`, when it stores a byte successfully, involves to a symbolic constraint where the immediate argument `alignment` is a power of 2 and not greater than 4, and the addition of the second operand and the immediate argument `offset` does not exceed the memory size defined in the seed.

Additionally, inspired by the success of previous work [11], [14], [40], we also generate symbolic constraints where the operands are special values (e.g., 0 and 1 among others for an operand of type `i32`) to create more representative inputs.

Finally, during testing, given a selected seed, we determine concrete values for the input of the *target instruction* by solving the symbolic constraint using a Satisfiability Modulo Theory (SMT) solver. In total, we identify representative inputs for 424 instructions, covering all instructions except the control instructions (tested via the *Code Insertion Mutation*).

(3) Position-Aware Insertion. An instruction's position can influence its behavior. For example, in WAMR, the execution of the `local.set` instruction is affected by the control flow in which it resides, due to optimization; its execution result can be incorrect when it is within certain types of control flow. We therefore aim to generate test cases where the *target instruction* is placed at potentially vulnerable positions. To this end, we develop eight insertion strategies (see Table I), considering the control flow constructs and the control instruction `return`, to determine insertion positions.

We additionally design six *wrapping* methods (see Table II) that modify the instructions surrounding the target instructions, to diversify the control flow. For example, we wrap the *target instruction* `i32.add` in Fig. 5(a) with the wrapping method *Wrap Block*, leading to the instructions in Fig. 5(b).



```
i32.const 1
i32.const 1
i32.add           ;; target instruction
```

(a) The original `i32.add`

```
i32.const 1
i32.const 1
block (param i32 i32) (result i32)
      i32.add      ;; target instruction
end
```

(b) The wrapped `i32.add`

Fig. 5. An example of wrapping

Code Insertion Mutation inserts a code snippet with multiple instructions and complex control flows to generate test cases, testing whether the runtimes can handle such test cases. This entails mechanisms to (1) generate diverse control flow constructs and (2) ensure the validity of the generated test cases. We use a grammar-based approach, supplemented by the extracted specification knowledge, to achieve these goals. This

TABLE I

INSERTION STRATEGIES. BLOCK, LOOP, IF, AND ELSE ARE CONTROL FLOW CONSTRUCTS IN WASM.

Strategy	Description
Random	Insert the instructions into a random position.
Inner Block	Insert the instructions into a block.
Inner Loop	Insert the instructions into a loop.
Inner If	Insert the instructions into an if branch.
Inner Else	Insert the instructions into an else branch.
New Function	Insert the instructions into a new function.
Before Return	Insert the instructions before the first `return`.
After Return	Insert the instructions after the first `return`.

TABLE II

WRAPPING METHODS. BLOCK, LOOP, IF, AND ELSE ARE CONTROL FLOW CONSTRUCTS IN WASM.

Method	Description
No Wrap	Leave the *target instruction* unmodified.
Wrap Block	Wrap the *target instruction* inside a block.
Wrap Loop	Wrap the *target instruction* inside a loop.
Wrap If	Wrap the *target instruction* inside the if branch of a conditional construct.
Wrap Else	Wrap the *target instruction* inside the else branch of a conditional construct.
After Return	Add a random instruction sequence containing a `return` instruction before the *target instruction*.

process involves three steps. First, we determine the control flow constructs of the code snippet to be inserted. Second, we insert valid instructions into the control flow constructs to create a valid code snippet. Finally, we insert the code snippet into a seed to create a new test case.

(1) Grammar-based Control Flow Construct Generation. We determine the control flow constructs using a grammar (Fig. 6) summarized from the LLM-extracted knowledge control flow.

We use the first six rules in Fig. 6 to randomly generate control flow constructs. The first production rule generates a sequence of instructions indicating control flow constructs, and the $\langle \text{InstSeq}_P \rangle$ symbol, a placeholder for an instruction sequence. We assign a randomly generated type to each $\langle \text{InstSeq}_P \rangle$. We infer the associated types of the control flow constructs from the bottom up, according to the language specification (e.g., the type of a $\langle \text{Block} \rangle$ is the same as the type of the $\langle \text{InstSeq}_P \rangle$ it contains).

(2) Instruction Generation. We generate a valid instruction sequence with the required type to replace each $\langle \text{InstSeq}_P \rangle$, based on knowledge about instruction validation, one instruction at a time. We speed up sequence completion as it nears a preset length (10, in our implementation) by adding padding instructions to refine the sequence's type, eliminating excess operands, and incorporating necessary ones. We add two kinds of padding instructions to achieve this. First, if the generated instructions produce more operands than expected, we insert instructions to store these excess operands in the global variables, and Wasm-Smith [17] also takes this strategy. Second, if the generated instructions lack operands, we pad constant instructions (e.g., `i32.const`) to ensure the resulting instruction sequence matches the expected type.

$$\begin{array}{rcl}
\langle \text{Code} \rangle & ::= & \langle \text{InstSeq} \rangle \\
\langle \text{InstSeq} \rangle & ::= & \langle \text{loopBlock} \rangle \mid \langle \text{Block} \rangle \mid \langle \text{ifBlock} \rangle \\
 & & \mid \langle \text{InstSeq}_P \rangle \mid \langle \text{InstSeq} \rangle \langle \text{InstSeq} \rangle \\
\langle \text{ifBlock} \rangle & ::= & \text{if } \textit{blockType } \langle \text{InstSeq} \rangle \text{ end} \\
\langle \text{ifBlock} \rangle & ::= & \text{if } \textit{blockType } \langle \text{InstSeq} \rangle \\
 & & \text{else} \langle \text{InstSeq} \rangle \text{ end} \\
\langle \text{Block} \rangle & ::= & \text{block } \textit{blockType } \langle \text{InstSeq} \rangle \text{ end} \\
\langle \text{loopBlock} \rangle & ::= & \text{loop } \textit{blockType } \langle \text{InstSeq} \rangle \text{ end} \\
\langle \text{InstSeq}_P \rangle & ::= & \langle \text{inst}_P \rangle^* \\
\langle \text{inst}_P \rangle & ::= & \text{i32.const} \mid \text{f32.const} \mid \text{i32.add} \mid \ldots
\end{array}$$

Fig. 6. Grammar rules for code generation

(3) Code Insertion. To ensure that generated code can potentially affect the behavior of the original test case, we insert it into the *entry* function (i.e., the function to be executed). We only insert the generated code before instructions that could prematurely terminate the function's execution (e.g., a `return` instruction) to prevent the generated code from being skipped.

2) Module Definition Mutation: Module variables defined (e.g., linear memory) can also affect Wasm program behavior [19]. We therefore design a mutation operator to mutate test case definitions via deletion, insertion, or replacement. To construct valid definitions for insertion or replacement, we first use the LLM to identify the fields of a definition, and their corresponding types. We populate these fields with valid values to construct a new definition. For example, when generating a definition for a global variable, the LLM identifies its fields: value type, mutability, and constant expression. Value types include, for example, `i32`, `f32`; mutability is either *const* (encoded as 0) or *var* (encoded as 1); and constant expressions are constructed to match the value type, such as `i32.const 0` for an `i32`. Table III shows a subset of these mutations, of the 22 total.

3) Byte Mutation: The main goal of byte mutation is to generate malformed test cases to trigger decoding bugs. Failure to detect and abort on malformed test cases can lead to unexpected runtime behavior. For example, WAMR incorrectly decodes the nonsense bytes `0xFD8C` into the instruction `i16x8.shr_s`. We propose two mutation operators to generate test cases containing malformed functions and malformed sections, respectively:

Malformed Function Mutation. To test whether the runtimes decode Wasm functions correctly, we mutate bytes encoding the *entry* function in two ways: (1) *Function body mutation*. We randomly insert, replace, or delete a byte in those corresponding to the entry function. (2) *Function size declaration mutation*. Since function size (i.e., the number of bytes encoding the function) must be declared at its start, we

TABLE III
PARTIAL MUTATIONS FOR MODULE DEFINITION.

Mutation	Description
Insert Table	Insert a randomly generated table.
Insert Memory	If the seed does not define a linear memory, insert one.
Replace Memory	Replace the linear memory with a randomly generated one.
Insert Passive Data Segment	Insert a randomly generated passive data segment.
Insert Active Element Segment	Insert a randomly generated active element segment.
Insert Function	Insert a randomly generated function.
Insert Global Variable	Insert a global variable.

test the runtime's ability to detect violations by deliberately mismatching the declared size with the actual length, with a small probability.

Malformed Section Mutation. To test whether the runtime can detect malformed module definitions (e.g., global variables) in other sections (e.g., the global section) of a Wasm program, we implement a two-step mutation operator to mutate bytes encoding these sections. First, it randomly selects a section for mutation. Then, similar to mutating the bytes encoding the *entry* function, it mutates the bytes encoding the selected section, and the bytes declaring the section size.

C. Mutation Selector

Our mutation operators can generate diverse test cases. However, scheduling these mutation operators to explore the runtime code space efficiently is still a challenge. A straightforward scheduling method is to sample the operators uniformly at random. However, this approach fails to evaluate operator effectiveness, and tends to waste computation on ineffective operators [20], [24].

Inspired by recent advancements in online learning [41], we instead frame the testing process as an online game. In each round, LWDIFF uses historical testing results to inform operator selection. We model mutation operator selection as a multi-armed bandit (MAB) problem [26], wherein each mutation operator is an "arm" that, when selected, yields a distinct and previously unknown distribution of rewards. In the testing context, the "reward" corresponds to mutation operator efficacy. Therefore, our objective is to maximize the number of bugs uncovered within a finite testing period by identifying and prioritizing the mutation operators that have the highest potential to uncover bugs.

An effective reward mechanism is crucial for guiding selection. Branch coverage can serve as a proxy for testing effectiveness and is widely used by testing algorithms for similar purposes [22], [42]. Therefore, we similarly use branch coverage as a reward mechanism to encourage the generation of test cases that cover new code. We adopt the UCB1-Tuned algorithm [35], which has been proven effective in multiple testing approaches [21], [43], to facilitate mutation operator scheduling. UCB1-Tuned considers historical rewards and selects a mutation operator as:

$$MO_{idx}(t) = \arg\max_{a} \left(\bar{x}_a + \sqrt{\frac{\ln t}{n_a} \min\left(\frac{1}{4}, V_a(n_a)\right)} \right) \tag{1}$$

$$V_a(n_a) = \sigma_a^2 + \sqrt{\frac{2\ln t}{n_a}} \tag{2}$$

In this formula, $MO_{idx}(t)$ refers to the mutation operator selected during the t_{th} selection. The term $\bar{x}_a$ refers to the average reward of the a_{th} mutation operator, n_a refers to the number of times that the a_{th} mutation operator has been selected, and σ_a^2 refers to the sample variance of the rewards for the a_{th} mutation operator. When selecting the next mutation operator, we calculate the reward according to the new branches triggered by the last mutation operator and update the historical rewards. Then, we determine the optimal mutation selector according to Equation 1.

D. Differential Testing Engine

The differential testing engine sends a test case to different runtimes for execution, collects the output from the runtimes, and identifies whether there are differences in those outcomes, indicating a potential bug. To achieve this, we first determine the variables to model differences between runtimes. After executing a test case, we collect these variables, and compare them, to detect potential differences. Finally, given the potentially large number of differential test cases, we use a keyword-based method over the runtime exception methods introduced in WADIFF [11] to de-duplicate the differential outcomes and tests.

1) Difference Detection: We represent execution results, denoted as S, by a combination of $\{Return, Exec\}$. $Return$ refers to the return value of the *entry* function of the test case. $Exec$ refers to a runtime execution status, i.e., whether the execution leads to an exception. Given n runtimes under test, and the execution result of i_{th} runtime denoted as S_i, we say there is a difference between them iff:

$$\exists i, j \in \{0, 1, \ldots, n-1\}, S_i(tc).\lambda \neq S_j(tc).\lambda$$

where λ refers to either the return value or the execution status of the runtime. If any two S contain different return values or execution statuses, we say the test is differentiating.

2) Variable Collection: We collect variables for difference detection after the *entry* function executes. We straightforwardly instrument the runtime to collect the return value of the entry function. Meanwhile, execution can result in one of four execution statuses: whether it triggers an exception (e.g., caused by an invalid test case), leads to a runtime crash, fails to terminate within the predefined time limit (one second, in our implementation), or the runtime executes the test case successfully.

3) Keyword-based Clustering: The goal of keyword-based clustering following the prior work [11] is to de-duplicate the differences identified in the previous step. We characterize the root cause of the difference from three failed executions: crashes, timeouts, and exceptions. For exceptions, we predefine the keywords to string-mapping the error messages, e.g., "divided by zero" and "illegal opcode", to identify these root causes. Test cases are clustered when they have the same execution status and share the same failure root cause. We then sample at least one case from each cluster for analysis.

E. Seed Pool Update

Since mutating promising test cases can lead to more promising test cases [23], we identify such test cases and use them as seeds to generate new test cases, facilitating the testing process. For each test case, we first determine if it can be an effective seed; if so, we then decide whether to add it to the seed pool.

Effective Seed. We consider a test case that can be executed across multiple runtimes without exceptions and leads to finite executions as qualifying as an effective seed. If a test case triggers an exception on a runtime, its derived test cases will likely trigger the same exception [11]. When a test case (denoted as TC_E) triggers an exception on all the runtimes, it results in no differences. Additionally, the derived test cases of TC_E are also likely to trigger exceptions on all runtimes, resulting in no differences. Although a mutation can potentially lead to a difference (e.g., inserting an instruction triggering a bug), if it is applied to a test case (e.g., TC_E) that triggers exceptions on all runtimes, no differences are observed, leading to inefficient testing. Test cases that result in non-terminating executions are excluded, for efficiency.

Seed Inclusion Criteria. For a test case that qualifies as an effective seed, if it can trigger unexplored code, we add it to the seed pool. Efficient fuzzing is achieved by test cases that broaden or introduce new code coverage [22], [25], [23]. Otherwise, we will also randomly add a qualifying test case to the seed pool with a small probability (0.003, in our implementation).

IV. IMPLEMENTATION

Runtime for Coverage Collection. We use WAMR's JIT mode [28] as our reference runtime for code coverage collection, for two reasons. First, WAMR's implementation in C allows for lightweight branch coverage collection with LLVM. Second, WAMR's JIT mode offers comprehensive support for almost all Wasm language features, meaning it can execute a wide diversity of generated tests.

LLM Setup. Our implementation is built on GPT-4, accessed through the API. We set the temperature parameter to zero. We justify the design choice to modulate the randomness and creativity of the LLM to mitigate the risk of hallucination, following prior work [40]. We set the maximum number of retries to 3 for the interactive prompt refinement. To retrieve specification knowledge for LLM, we build a knowledge retriever by parsing the Wasm specification with the help of Docutils [39]. In our implementation, we combine the four instruction-related tasks into a single prompt, enabling the LLM to complete all tasks at once. This avoids the inefficiency of using separate prompts, which would require repeatedly providing the same background information, wasting tokens.

V. EVALUATION

In this section, we aim to answer the following questions:

- **RQ1 (effectiveness)**: How effective is LWDIFF in terms of code coverage and bug detection, compared with baseline methods?
- **RQ2 (ablation study)**: What is the effectiveness of LLM-extracted specification knowledge, position-aware insertion, and MAB-based mutation selector for guiding testing?
- **RQ3 (qualitative analysis)**: What are the root causes of the identified bugs in Wasm runtimes?

A. Setup

Metrics. We employ two metrics to evaluate Wasm testing tools: code coverage, and detected bug count. A testing approach is considered more efficient if it is able to trigger a wider range of code, and uncover a higher number of bugs.

Baselines. To the best of our knowledge, there are four closely related approaches to our own: Wasm-Smith [17], WADIFF [11], WASMaker [16], and Wapplique [44]. Our criteria for selecting baseline methods were: (a) they should target Wasm, and (b) they should be open-sourced. We selected two state-of-the-art methods, Wasm-Smith [17] and WADIFF [11], as baselines. WASMaker [16] and Wapplique [44] were excluded because they were not available open-source at the time of our experiments. Wasm-Smith [17] is a grammar-based test case generator. To evaluate Wasm-Smith, we collect a sufficient number of test cases generated by Wasm-Smith before testing, and forward them to the differential testing engine until the specified time limit expires. WADIFF [11] generates test cases in two steps. First, it uses a symbolic execution engine to generate representative test cases for each instruction. Then, it uses a byte mutator to mutate these test cases and generate new ones.

Runtimes under Test. We conduct evaluations on representative popular, well-maintained Wasm runtimes. We select runtimes with more than 4K stars on GitHub that have been maintained in the last year. Therefore, we select four projects: Wasmer [27], WasmEdge [6], WAMR [28], and Wasmtime [4]. Furthermore, we compile WAMR to five modes: classic interpreter mode, fast interpreter mode, JIT mode, fast JIT mode, and multiple-tier JIT mode. In total, there are eight runtimes under test.

Initial Seed. For reproducibility, we keep the number of initial seeds low in our evaluation [24]. We initialize the seed pool with a single simple test case that contains only one `nop` instruction. Note that the users can customize the seed pool.

B. RQ1: LWDIFF Effectiveness

To investigate LWDIFF ability to identify runtime bugs, we conduct differential testing over a span of 24 hours across eight runtimes, and compare code coverage and detected bug count with that achieved by existing tools. In 24 hours, LWDIFF generates 262,531 test cases triggering differences among runtimes. By analyzing these differences, we locate 31 bugs. Table IV (columns 1-3) shows results. Notably, LWDIFF triggers 31 bugs, and achieved the highest code coverage of 31.02%, significantly outperforming the baselines WADIFF [11] and Wasm-Smith [17], which trigger 6 and 15 bugs, respectively. Compared to the baselines, our approach improves code coverage by 22.17% (from 25.39% to 31.02%) and increases the number of detected bugs by 106.7% (from 15 to 31). Importantly, all 31 bugs are confirmed and fixed, and 25 of them are zero-day vulnerabilities.

Note that coverage results are most informative comparatively (across baselines), rather than in raw terms. First, each binary includes components we do not aim to cover. For example, WAMR includes code implementing the C API that we don't expect our generated tests to execute, which decreases overall coverage. Second, our experiments also only exercise one of the available runtime modes that a given binary implements. For example, we use WAMR's JIT mode to collect code coverage in experiments. We therefore do not cover the code implementing, e.g., the classic interpreter mode.

Table V details the number of bugs detected in each runtime. Note that a single bug may appear in multiple runtimes. Two bugs are found in WasmEdge, while the remaining bugs are discovered in various WAMR modes. The highest number of bugs, 23 in total, are found in WAMR's fast interpreter mode (WAMR-FINTERP), highlighting the vulnerabilities of this runtime mode in particular.

TABLE IV
CODE COVERED, AND BUGS DETECTED BY LWDIFF AND BASELINES IN ONE RUN.

	Coverage	Bug Count
WADIFF	25.39%	6
Wasm-Smith	21.03%	15
LWDIFF	**31.02%**	**31**
LWDIFF-RM	16.06%	6
LWDIFF-RS	30.89%	28
LWDIFF-RP	30.25%	25

LWDIFF outperforms WADIFF for two reasons. First, LWDIFF can generate test cases containing diverse control flow. WADIFF cannot, and therefore its generated tests do not trigger code that handles control flow. Meanwhile, LWDIFF applies the position-aware insertion to trigger the instruction behaviors determined by the instruction position, while WADIFF cannot achieve this and fails to test the corresponding code space comprehensively. Compared to Wasm-Smith, LWDIFF can test the decoding, validation, and execution phases, while Wasm-Smith only generates valid test cases for the execution phase. This allows us to trigger more code and bugs in the decoding and validation phases.

> **Answer to RQ1:** LWDIFF achieves the highest code coverage (31.02%) and identifies the largest number of bugs (31) in real-world wasm runtimes in 24 hours. Compared with the baseline method, LWDIFF increases the code coverage by 22.17% and the number of detected bugs by 106.67%.

TABLE V
DISTRIBUTION OF 31 DETECTED WASM RUNTIME BUGS. THE SUFFIXES OF THE RUNTIMES REPRESENT DIFFERENT MODES OF WAMR. FOR EXAMPLE, WAMR-JIT REPRESENTS WAMR'S JIT MODE.

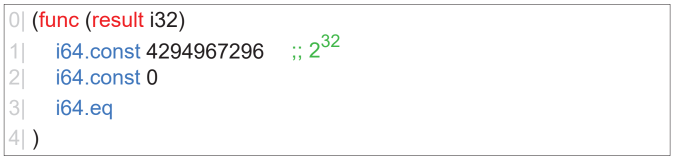
Runtime	(New) Bugs Found
WasmEdge	2
WAMR-FINTERP	23
WAMR-JIT	11
WAMR-MJIT	10
WAMR-FJIT	9
WAMR-CINTERP	7
Total(unique)	31

C. RQ2: Ablation Study

Our previous results show that LWDIFF overall outperforms the baselines. In this section, we perform an ablation study to evaluate the degree to which each component of LWDIFF— LLM-extracted specification knowledge, position-aware insertion, and the mutation selector — contributes to its success in terms of both code coverage and bug detection. To achieve this, we design the following baselines.

- LWDIFF: The complete tool includes all mutation operators, position-aware insertion, and the MAB-based mutation selector.
- LWDIFF-RM: This baseline excludes the LLM-assisted mutation operators and includes only byte mutation operators. This baseline allows us to verify whether LLM-assisted mutation operators enhance testing effectiveness.
- LWDIFF-RP: This baseline includes a *Target Instruction Insertion Mutation* that does not employ position-aware insertion. Instead, it inserts the *target instruction* at a random position without wrapping it. We design the baseline to verify whether position-aware insertion can improve testing effectiveness.
- LWDIFF-RS: This baseline employs a random mutation selector, which samples mutation operators with equal probability. We build this baseline to verify whether the MAB-based mutation selector improves testing efficiency.

Results. We run each baseline for 24 hours, and then compare the code coverage and detected bug count. We summarize the code coverage and detected bug count for each baseline in Table IV (columns 3-6). LWDIFF achieves the highest code coverage (31.02%) and detects the largest number of



```
(func (result i32)
    i64.const 4294967296    ;; 2^32
    i64.const 0
    i64.eq
)
```

Fig. 7. A function triggers a bug in instruction implementation

bugs. Since LWDiff-RM does not use the LLM-assisted mutation operators, it can hardly generate well-formatted test cases, making it difficult to trigger the runtime's code for the validation and execution phases. It achieves the lowest code coverage (16.06%) and only triggers six bugs, indicating that LLM-assisted mutation operators significantly improve testing effectiveness. LWDiff-RP achieves a lower code coverage (30.25%) than LWDiff, speaking to the importance of position-aware instruction insertion strategy (rather than a random strategy). LWDiff-RS also achieves lower code coverage, highlighting the effectiveness of the MAB-based mutation selector.

> **Answer to RQ2:** LLM-assisted mutation operators, the MAB-based mutation selector, and the position-aware insertion all show positive impacts on code coverage and bug detection. Among them, the LLM-assisted mutation operators achieve the largest boost.

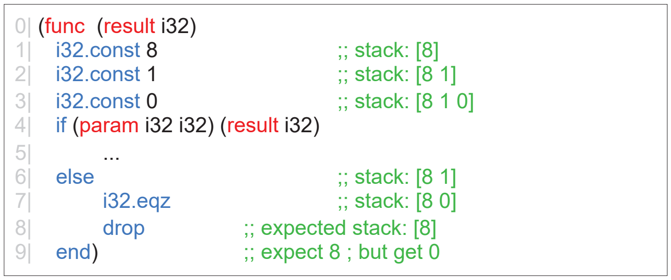
D. RQ3: Root Cause Analysis

We manually analyze all 31 bugs detected by LWDiff, identifying three primary root causes:

Incorrect Decoding. We identify that seven out of 31 bugs arise due to the incorrect implementation of the runtime's decoding phase. Six of these are newly-discovered bugs. By analyzing the root causes of these bugs, we found that a Wasm runtime can fail to detect and raise exceptions for test cases with undefined opcodes, undefined operand types, or illegally encoded control flow. For instance, we found a unique bug where all the WAMR runtimes under test failed to raise an exception for a test case with an illegally encoded control flow. In another bug of this type, WasmEdge failed to detect and terminate a test case containing illegally encoded local variable definitions.

Incorrect Instruction Implementation. We find that seven out of 31 bugs arise due to an incorrect implementation of instruction validation or execution. Six of these bugs are newly-discovered by LWDiff.

Fig. 7 illustrates such a bug. The instruction `i64.eq` is designed to take two operands as input, output 1 if the operands are equal, and 0 otherwise. In this case, the instruction `i64.eq` takes two operands $4, 294, 967, 296$ (i.e., 2^{32}) and 0; the expected output should be 0. However, in WAMR's fast JIT mode, the actual output of `i64.eq` is 1, because the code uses 32-bit variables to store 64-bit operands.



```
0| (func  (result i32)
1|    i32.const 8                    ;; stack: [8]
2|    i32.const 1                    ;; stack: [8 1]
3|    i32.const 0                    ;; stack: [8 1 0]
4|    if (param i32 i32) (result i32)
5|          ...
6|    else                           ;; stack: [8 1]
7|          i32.eqz                  ;; stack: [8 0]
8|          drop             ;; expected stack: [8]
9|    end)                   ;; expect 8 ; but get 0
```

Fig. 8. A function triggers a bug in control flow handling

Incorrect Control Flow Handling. We identify that 11 of 31 bugs arise because the runtime incorrectly implements control flow semantics, leading to incorrect outputs and even crashes. We discovered eight of them for the first time.

For example, Fig. 8 illustrates such a bug found in WAMR's fast interpreter mode. An if-else block (lines 4-9) takes three operands: 8, 1, and 0. The else branch is executed because the control flow of the if-else block is determined by the top operand on the stack (i.e., 0, determined by line 3). In the else branch, the instruction `i32.eqz` takes the operand 1 and pushes 0 to the stack (line 7), as `i32.eqz` outputs 0 if the input is not 0. Then, the output of `i32.eqz`, i.e., 0, is popped by the instruction `drop` on line 8. Therefore, at the end of the else branch, 8 is expected to be left on the stack, which is also the function's return value. However, due to a bug in stack management, the output of `i32.eqz` is not removed, and the return value is 0.

Remaining Bugs. The remaining bugs stem from various root causes. For example, in WAMR's fast interpreter mode, there is a bug where an `i64` type variable is copied incorrectly. This mistake can cause the high 32 bits of the newly copied variable to potentially overlap with the low 32 bits. For instance, if the original variable is 1, the erroneously copied new variable turns into $0x100000001$.

> **Answer to RQ3:** Across the 31 detected bugs in real-world Wasm runtimes, our manual analysis identifies three major causes: incorrect decoding (seven bugs), incorrect instruction implementation (seven bugs), and incorrect control flow handling (11 bugs).

VI. Discussion and Threats to Validity

A. Robustness Analysis

One risk to the validity of our experiments is the effect of randomness on our results. To mitigate and evaluate this risk, we run all baselines 10 times, and show the statistics in Table VI. The first column gives the baselines; the second column, average branch coverage. We assess statistical significance using p-values, and quantify the effect size of differences between baselines and LWDiff, with the A_{12} metric, as shown in the third and fourth columns, respectively.

We make the following observations from Table VI: P-values for all comparisons, except for LWDiff-RS, are below

TABLE VI
AVERAGE CODE COVERAGE IS REPORTED ALONGSIDE STATISTICAL ANALYSES, INCLUDING P-VALUES (MANN-WHITNEY U-TEST) AND VARGHA-DELANEY EFFECT SIZES (A_{12}).

	Avg. Coverage (10 runs)	p-value	A_{12}
LWDIFF	**31.18%**	-	-
WADIFF	25.63%	1.81e-4	1.00
Wasm-Smith	21.02%	1.80e-4	1.00
LWDIFF-RM	16.14%	1.82e-4	1.00
LWDIFF-RS	31.13%	0.088	0.73
LWDIFF-RP	31.05%	7.30e-4	0.95

the commonly used threshold of 0.05, suggesting statistical significance, and providing evidence against the null hypothesis of no difference in code coverage between LWDIFF and the baselines. Regarding effective size statistics, the A_{12} values for all comparisons exceed the widely used threshold of 0.7, reflecting an advantage of LWDIFF over the baselines in terms of code coverage. The exception is LWDIFF-RS where the p-value (0.088) slightly exceeds the 0.05 threshold. However, the effect size (A_{12}=0.73) still suggests that LWDIFF maintains an advantage compared to the baseline. These results provide evidence that the improvement of LWDIFF over the baselines is consistent, and should not be attributed to chance.

B. Correctness Analysis of the Extracted Specifications

Our approach relies heavily on an LLM extracting information about a language specification. We analyzed both syntactic and semantic correctness of the extracted specifications. For module definitions, 12 out of 13 responses (92.3%) are well-formatted. For instructions, the syntactic correctness for four instruction-related tasks, including instruction format, instruction type, instruction validation rules, and instruction execution behavior conditions, achieves accuracies of 99.1%, 99.5%, 99.5%, and 99.3%, respectively. For control flow constructs in Wasm, the LLM successfully identifies all three constructs with the required format, providing a reliable starting point for a later manual adjustment.

We manually evaluated the semantic correctness of above identified syntactically correct specifications, focusing on module definitions and instructions. For module definitions, our analysis shows that 7 out of 12 well-formatted responses are semantically correct, an accuracy of 58.3%. Additionally, we assessed semantic correctness for three instruction-related tasks, including instruction format, instruction type, and instruction validation rules, achieving accuracies of 96.7%, 96.7%, and 88.4%, respectively.

Overall, our analysis demonstrates a high level of syntactic and semantic correctness in using the LLM to extract specifications.

C. External Threats to Validity

The differential testing engine of LWDIFF requires instrumenting the runtimes under test to detect differences among them, necessitating some manual effort. Fortunately, this effort is manageable. In our evaluation, we instrument the runtimes from four different projects using fewer than 400 lines of code in total, which we consider acceptable.

VII. RELATED WORK

There are multiple prior works proposed for testing Wasm runtimes, including Wasm-Smith [17], WADIFF [11], Wasm-Fuzzer [18], and WASMaker [16]. However, they all suffer from some limitations and none of these can generate diverse test cases covering all three phases. Wasm-Smith [17] is a test case generator for Wasm. It employs a grammar-based approach to generate instructions randomly, ensuring syntactic correctness while considering stack balance. However, because it assigns the instruction inputs randomly, it cannot comprehensively test the implementation of instructions. Furthermore, unlike LWDIFF, it lacks a bug oracle to detect bugs. WADIFF [11] is a differential testing framework for Wasm runtimes. It uses manually defined patterns to extract the execution behavior of each instruction and employs this knowledge to generate test cases. However, it has two limitations compared to LWDIFF. First, it can hardly generate test cases with complex control flow constructs and test whether the runtimes can handle them. Second, it cannot comprehensively test the implementation of instructions because it does not consider the position of the instruction. WasmFuzzer [18] is a fuzzing framework for testing Wasm runtimes. However, its naive mutators struggle to generate test cases with diverse code. WASMaker [16] is a differential testing framework that generates test cases by AST mutation. Compared to LWDIFF, it has three main limitations. First, it does not generate representative inputs for instructions to test the implementation of instructions comprehensively. Second, it does not generate illegally encoded test cases, thereby failing to detect bugs in the decoding phase. Third, it cannot test module definitions as comprehensively as LWDIFF.

VIII. CONCLUSION

In this work, we presented LWDIFF, a differential testing framework for Wasm runtimes. LWDIFF uses the LLM to extract the knowledge from the specification and uses such knowledge to build mutation operators that can generate diverse test cases. Through extensive evaluation, LWDIFF demonstrated superior performance in both code coverage and bug detection compared to state-of-the-art methods. Specifically, LWDIFF successfully identified 31 bugs across eight popular Wasm runtimes, with 25 of these bugs being previously undiscovered.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This research is partially supported by Hong Kong RGC Project (PolyU15224121) and Hong Kong ITF Project (PRP/005/23FX).

REFERENCES

[1] "The main page of eosio," https://eos.io/, 2024.

[2] "The main page of near," https://near.org/, 2024.

[3] "The main page of coreum," https://www.coreum.com/, 2024.

[4] "The main page of wasmtime," https://wasmtime.dev/, 2024.

[5] B. L. Titzer, "A fast in-place interpreter for webassembly," *arXiv preprint arXiv:2205.01183*, 2022.

[6] "The main page of wasmedge," https://wasmedge.org/, 2024.

[7] 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 *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1986–2001.

[8] "The main page of wasm3," https://github.com/wasm3/wasm3, 2023.

[9] "A vulnerability in cosmwasm," https://github.com/CosmWasm/advisories/blob/main/CWAs/CWA-2023-004.md, 2023.

[10] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting webassembly runtime bugs," *arXiv preprint arXiv:2301.12102*, 2023.

[11] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "Wadiff: A differential testing framework for webassembly runtimes," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 939–950.

[12] G. Hamidy *et al.*, "Differential fuzzing the webassembly," 2020.

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

[14] M. Jiang, T. Xu, Y. Zhou, Y. Hu, M. Zhong, L. Wu, X. Luo, and K. Ren, "Examiner: automatically locating inconsistent instructions between real devices and cpu emulators for arm," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 846–858.

[15] S. Li and Z. Su, "Ubfuzz: Finding bugs in sanitizer implementations," *arXiv preprint arXiv:2401.04538*, 2024.

[16] 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*, 2024, pp. 1262–1273.

[17] "Wasm-smith," https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith, 2023.

[18] 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.

[19] "The main page of webassembly.org," https://webassembly.org/, 2024.

[20] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[21] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1634–1645.

[22] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, 2023.

[23] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[24] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "Darwin: Survival of the fittest fuzzing mutators," *arXiv preprint arXiv:2210.11783*, 2022.

[25] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[26] A. Mahajan and D. Teneketzis, "Multi-armed bandit problems," in *Foundations and applications of sensor management*. Springer, 2008, pp. 121–151.

[27] "The main page of wasmer," https://github.com/wasmerio/wasmer/, 2024.

[28] "The main page of webassembly micro runtime," https://bytecodealliance.github.io/wamr.dev/, 2024.

[29] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, "Exploring the use of webassembly in hpc," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 92–106.

[30] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2020, pp. 1–4.

[31] 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*, 2017, pp. 185–200.

[32] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–41, 2021.

[33] "The main page of wasmvm," https://github.com/CosmWasm/wasmvm, 2024.

[34] "The github main page of zkwasm," https://github.com/DelphinusLab/zkWasm, 2024.

[35] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[36] S. Hwang, S. Lee, J. Kim, and S. Ryu, "Justgen: effective test generation for unspecified jni behaviors on jvms," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1708–1718.

[37] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+ 1-version differential testing of both javascript engines and specification," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 13–24.

[38] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[39] Docutils Project, "Docutils: Documentation utilities," https://docutils.sourceforge.io/, 2024.

[40] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 196–196. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00211

[41] A. Slivkins *et al.*, "Introduction to multi-armed bandits," *Foundations and Trends® in Machine Learning*, vol. 12, no. 1-2, pp. 1–286, 2019.

[42] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[43] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," 2021.

[44] 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*, 2024, pp. 1035–1047.