2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00188

WADIFF: A Differential Testing Framework for WebAssembly Runtimes

Shiyao Zhou[1], Muhui Jiang[1*], Weimin Chen[1], Hao Zhou[1], Haoyu Wang[2], Xiapu Luo[1]
[1] The Hong Kong Polytechnic University, Hong Kong, China
[2] Huazhong University of Science and Technology , Wuhan, China
{csszhou1, cswchen, cshaoz, csxluo}@comp.polyu.edu.hk, muhjiang@polyu.edu.hk, haoyuwang@hust.edu.cn

***Abstract*—WebAssembly (Wasm) runtime provides a virtual machine that can execute the WebAssembly modules and is widely used in different areas (e.g., browsers, edge computing, blockchain). Thus, the precision and reliability of the WebAssembly runtime are important and deserve our attention. To ensure the correctness and detect potential bugs in WebAssembly runtimes, we propose WADIFF, a differential testing framework, which consists of a sufficient test case generator and a deterministic differential testing engine. To evaluate the effectiveness of WADIFF, we apply it to seven popular WebAssembly runtimes and found 417 inconsistent instructions due to bugs and different implementations in the runtimes. Furthermore, we identify 21 bugs from 7 WebAssembly runtimes, and 8 of them are confirmed by their developers.**

***Index Terms*—WebAssembly, Differential Testing**

I. INTRODUCTION

WebAssembly is widely used in different areas (e.g., smart contracts [2], [3], IoTs [29], [30]) due to the high performance, sandboxed environment, and scalability in different OSes [34]. To execute the WebAssembly programs, WebAssembly runtimes, which decode, validate, and execute the WebAssembly bytecode, are required. To provide a comprehensive and detailed description of how the WebAssembly programs are executed, WebAssembly specification is proposed [4], which allows developers to implement different WebAssembly runtimes in a consistent and reliable manner.

In the event of any flaws or vulnerabilities in the current WebAssembly runtimes, the resulting execution may yield incorrect outcomes [17]. This can have significant consequences, particularly for blockchain systems that rely on smart contracts compiled into WebAssembly bytecode. Though some approaches that utilize fuzzing or differential testing techniques [21], [24], [18] are proposed to evaluate WebAssembly runtimes, they suffer from two common limitations. First, the illegal test cases, which cannot pass the validation process, are not well tested against the target runtimes. Furthermore, the legal test cases, which will be executed by the runtimes, are usually generated randomly and are not sufficient to explore the complex implementations of different runtimes. This leads to many bugs unexplored. Second, existing works do not model the state of WebAssembly runtime systematically. In this case, the semantic-related bugs, which can result in different runtime states without a crash, may not be identified.

* Corresponding Author.

To bridge this research gap, we propose a differential testing framework to detect the implementation flaws of WebAssembly runtimes. This is not easy and has two main challenges. The first challenge is how to generate sufficient test cases, which should cover both legal and illegal ones to test all the functionalities (i.e., decode, validate, and execute) of the runtimes. Furthermore, for legal test cases, they should cover as many execution behaviors described in the WebAssembly specification as they can. This can help to explore the potential bugs that are hard to be triggered by previous works. The second challenge is how to build a deterministic differential testing engine. Meanwhile, it is non-trivial to determine which attributes are required for comparison, which needs a systematic study on the WebAssembly runtimes, and how to dump these attributes. To address these challenges, we propose WADIFF, a differential testing framework for WebAssembly runtimes. The framework consists of a sufficient test case generator (§ IV) and a deterministic differential testing engine (§ V).

Sufficient Test Case Generator. We utilize the information provided by WebAssembly specification to generate test cases. Specifically, to analyze the semantics in the specification automatically, we design a DSL transformer to transform the semantics in natural language to a structured DSL. Based on the transformed DSL, we design a symbolic execution engine and utilize it to generate test cases that can explore execution behaviors described in the specification. To further generate test cases with illegal encoding, we feed the legal ones to a designated mutator with different mutation strategies.

Deterministic Differential Testing Engine. The differential testing engine receives the generated test cases as input. We insert prologue instructions to set the pre-state. Note that all runtimes execute the same prologue instructions in one test case, resulting in the same pre-state. Meanwhile, we systematically model the state of the WebAssembly runtime to help determine which attributes should be dumped. We instrument into the runtimes to dump the required attributes. If there is a difference between the dumped attributes, an inconsistent instruction is identified.

We implement WADIFF and apply it on 7 runtimes, including wasmer [10], wasmi [11], WAMR [13] in classic interpreter mode (denoted as WAMR-classic) and fast interpreter mode (denoted as WAMR-fast), wasm3 [8], WasmEdge [9], and WAVM [12]. In total, we generate 1,395,091 test cases and

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00188

124,157 of them trigger differences among the runtimes. We found differences triggered on 417 instructions and they are caused by various factors such as different supported features and implementation bugs. Furthermore, we identify 21 bugs, and 8 of them are confirmed or fixed.

The main contributions of this paper are as follows.

- **WebAssembly Specification Transformer** We propose a WebAssembly specification transformer that transforms the original natural language into our designed domain specific language (DSL). Based on the transformed DSL, different program analysis techniques can be applied.
- **Sufficient Test Case Generator** We build a symbolic execution engine based on the transformed DSL and a mutator for WebAssembly programs. With the symbolic execution engine and the mutator, we can generate both illegal and legal test cases.
- **Extensive Model of WebAssembly Runtimes** We study the WebAssembly runtimes systematically and model the runtime state with different attributes. Based on the model, we build a deterministic differential testing engine.
- **Effective Differential Testing Framework** We build an effective differential testing framework named WADIFF that consists of a sufficient test case generator and a deterministic differential testing engine. Our experiments show that WADIFF is general and can find bugs in the real world. To engage the community, we also open source WADIFF at https://github.com/erxiaozhou/WaDiff for further research.
- **Comprehensive Evaluation** We apply WADIFF on 7 popular WebAssembly runtimes. The result shows that we can generate sufficient test cases as expected and detect bugs in the wild. In total, 21 bugs are located in all the tested runtimes, with 8 of them being fixed or confirmed by the developers.

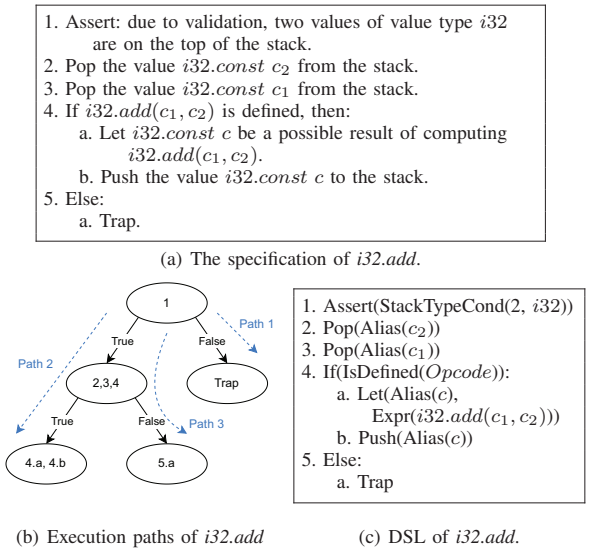
II. BACKGROUND AND MOTIVATION

A. WebAssembly Runtime

WebAssembly runtimes refer to the execution engines for WebAssembly bytecode. A WebAssembly execution procedure consists of three phases, including decoding, validation, and executions [4]. In the decoding phase, the WebAssembly bytecode is translated to a module. In the validation phase, the WebAssembly runtime checks whether the module is valid. If the module is invalid, the execution is terminated. Otherwise, the runtime instantiates the module and creates a module instance (§ II-B). Then, the runtime invokes the start function [4], which is the function specified via the user or the function with a specific name. A WebAssembly function consists of a sequence of instructions. Each instruction, which consists of an opcode and optional immediate arguments, takes operands from the operand stack (stack for short), performs its specific functionality, and pushes the result to the stack.

B. Module Instance

A module instance contains variables (e.g., linear memory, stack, functions, global variables, and the table), which an instruction interacts with. These variables influence the execution behaviors of an instruction because an instruction performs its functionality by interacting with variables in the instance. For example, the instruction *i32.store* takes two operands from the stack and stores an operand of type $i32$ in the linear memory. A difference between these variables of two instances can lead to different execution behaviors.



(a) The specification of *i32.add*.

(b) Execution paths of *i32.add*

(c) DSL of *i32.add*.

Fig. 1: The specification and execution paths of *i32.add*.

C. Motivating Example

We illustrate the workflow of WADIFF on testing the WebAssembly runtimes with a motivation example. Fig. 1(a) shows the specification describing the execution behavior of the instruction *i32.add*. There are three execution paths, which are shown in Fig. 1(b). The execution flow is determined by the constraints in lines 1 and 4.

Path 1. If the operands taken from the stack fail the assertion in line 1, i.e., the operands are not of type $i32$ (e.g., $f32$), the execution will be terminated.

Path 2. If there are two operands of type $i32$ on the stack top (line 1) and the opcode corresponds to *i32.add*, i.e., $0x6A$, the sum of two operands is pushed to the stack (lines 4-4.b).

Path 3. If there are two operands of type $i32$ on the stack top (line 1) while the opcode is undefined (line 4), the execution will be terminated (line 5.a).

1) Test case generator: We aim to generate test cases that can cover all execution paths described in the specification. To achieve this, we first transform the specification in semi-structured natural language to a structured DSL (§ IV-A). After the transformation, the specification of *i32.add* is represented in Fig. 1(c). To ensure all the execution paths can be covered, we design a symbolic execution engine (§ IV-B), which receives the specification in DSL as the input and generate test cases by solving the constraints.

For example, when generating cases for the instruction *i32.add*, which contains three execution paths, we determine the opcode and operands by solving the constraints on each path. Fig. 2(a), Fig. 2(b), and Fig. 2(c) show three test cases generated by solving the constraints on Path 1, Path 2, and Path 3, respectively. Furthermore, to test whether the decoding phase of a runtime can identify illegally encoded test cases as expected, we design a mutator (§ IV-C) to generate such test cases. For example, Fig. 2(d) shows a test case mutated from Fig. 2(b). The byte code denoting the function size is mutated from 07 to 06.

2) Differential testing engine: After the test cases are generated, we feed them into our differential testing engine. Note that the initial state is consistent for all runtimes. After the execution of the test cases, we dump the final state by instrumenting hooks in the target WebAssembly runtimes for comparison. A difference between the runtimes under test indicates a potential bug in the implementation. With the differential testing engine, we managed to detect three bugs, which are described below.

Case 1. Before executing a WebAssembly function, the runtime first decodes the test case according to the specification and only the function in a legal test case can be invoked. However, an illegal test case, which is encoded illegally, can be treated as legal by a wrongly implemented runtime. For example, the specification requires that the declared function size in the test case should equal the actual function size, or the test case should not pass the validation. Fig. 2(d) shows such a test case. However, due to a flaw in WasmEdge [9], the test case can pass the validation in WasmEdge, which is against the specification.

Case 2. According to the semantics indicated by Path 1, if the operands taken by the instruction *i32.add* are of unexpected types, the execution should be terminated. However, the test case in Fig. 2(a), where the instruction *i32.add* takes two operands of the type $i64$, can pass the validation in wasm3

```
07        // function size
...
42 05     // i64.const 5
42 10     // i64.const 10
A6        // i32.add
0B        // end
```

(a) A case for Path 1

```
07        // function size
...
41 05     // i32.const 5
41 10     // i32.const 10
A6        // i32.add
0B        // end
```

(b) A case for Path 2

```
07        // function size
...
41 05     // i32.const 5
41 10     // i32.const 10
C5        // Undefined opcode
0B        // end
```

(c) A case for Path 3

```
06        // function size
...
41 05     // i32.const 5
41 10     // i32.const 10
A6        // i32.add
0B        // end
```

(d) A case generated by mutator

Fig. 2: Code snippets of test cases. Each sub-figure displays the bytecode of the WebAssembly function on the left-hand side. The green-highlighted byte code indicates the opcodes of instructions, while the red-highlighted one in Fig. 2(d) represents the mutated byte code.

and be executed, which is an unexpected behavior.

Case 3. According to the semantics indicated by Path 3, if the opcode is undefined, the execution should be terminated. However, the case with the undefined opcode $C5$ (as shown in Fig. 2(c)) can be executed by WAMR-classic, causing an unexpected result.

Note that previous approaches may not detect these bugs as they cannot generate sufficient test cases that cover Path 2 and Path 3. Additionally, they may not generate the illegally encoded test cases that can trigger the unexpected behaviors (e.g., Case 1).

III. Methodology Overview

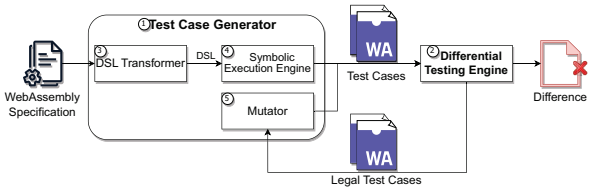


Fig. 3: The overview of WADIFF

As shown in Fig. 3, there are two components in WADIFF, i.e., a test case generator (①) and a differential testing engine (②). The test case generator will generate test cases (§ IV). Then, the differential testing engine executes these test cases, collects the state variables dumped from different WebAssembly runtimes, and detects the difference between runtimes (§ V). The differences between the state variables indicate potential bugs in WebAssembly runtimes.

A. Workflow

We design a transformer (③ in Fig. 3) to transfer the semi-structured specification in natural language to a structured DSL representation. Then, to generate representative test cases, we design a symbolic execution engine (④ in Fig. 3). Specifically, we first infer the symbolic constraints according to the DSL representation. Then, by solving the constraints on each path, we determine the candidates of immediate arguments and opcode that make up the instruction and the operands taken by the instruction. Furthermore, we also take test cases with illegal encoding into consideration. We design a mutator (⑤ in Fig. 3) to generate such test cases.

All test cases are sent to the ENGINE (§ V), which detects whether a test case causes a difference on different runtimes.

IV. Test Case Generator

The test case generator consists of three components and they are DSL transformer (§ IV-A), symbolic execution engine (§ IV-B), and mutator (§ IV-C).

A. DSL Transformer

To facilitate our further analysis, we propose a DSL for the WebAssembly specification and implement a transformer to convert the semi-structured natural language in specification into DSL (shown in Fig. 1(a) and Fig. 1(c)).

$$
\begin{aligned}
Description ::= & Statment* \\
Statment ::= & Pop|Push|Let|Execute|Trap|Replace|Invoke \\
& |Try|Return|Jump|Nop|Enter \\
& |If|Assert|Else|Repeat|While \\
Condition ::= & StackCond|EqualCond|OpDefinedCond \\
& |ExprCond|ExistCond|InstructionPartCond \\
& |CompareCond|SameTypeCond \\
Variable ::= & Imm|Opnd|Elem|Constant|Expr|Alias
\end{aligned}
$$

Fig. 4: DSL Grammar

Specifically, the DSL's grammar is shown in Fig. 4. In the DSL, we use a sequence of **Statements** (§ IV-A1) to describe the execution behaviors of an instruction in the original specification. The semantics of a Statement is determined by its type, as well as the **Variables** (§ IV-A3) and **Conditions** (§ IV-A2) it contains. We generate DSL in two steps. First, for an instruction, we transfer each sentence in the specification describing the instruction's execution behaviors to a DSL Statement and detect the type of each Statement with specified patterns. Second, we abstract the content in each sentence as Variables and Conditions in DSL.

1) Statement: Here we introduce how we transform the specification in natural language into Statements. In general, one line in the original specification maps to one Statement in DSL. We studied the original specification and classified the Statements into different types. For each type, we prepare the template for conversion. Fig. 5 shows an example, where "(.*)" represents a variable part and the other content is fixed. The sentence "Pop the value $i32.const\ i$ from the stack." will be transformed into a Pop Statement, because it matches the fixed part of the pattern. Because "the value $i32.const\ i$" matches the variable part of the pattern, it will be transformed into a Variable.

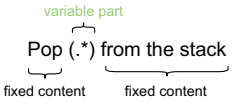


Fig. 5: A pattern example.

2) Condition: Conditions in DSL correspond to sentences in natural language that contain constraints. They usually follow certain keywords (e.g., "Assert", "If") and are related to the execution flow. To extract the Condition, we take a similar strategy and prepare the templates for conversion. For example, the conditional sentence in line 1 in Fig. 1(a), is transformed into a $StackCond(2, i32)$, where 2 means the number of operands to check and $i32$ means the expected operand type.

We categorize the Conditions into different types [5] according to the relationships they describe, facilitating the symbolic execution engine (§ IV-B) to construct constraints.

3) Variable: A Variable is extracted from nominal content in natural language (e.g., the object of a sentence), which refers to data. In practice, a Variable is extracted from the text matching the variable part of a pattern introduced in § IV-A1 and § IV-A2. For example, as shown in Fig. 5, for a sentence starting with "Pop" in the specification, the content between "Pop" and "from the stack" is transformed into a Variable.

To ensure that the Variable maintains the same semantics as the natural language it transformed from, we divide Variables into different types [14] according to the data a Variable refers to (e.g., an operand, a constant value). For each Variable, we abstract information from the natural language according to its type to describe its semantics. For example, we transform the text representing a $Constant$ value (e.g., 0) to a Constant Variable, whose semantics is determined by the constant value represented by the text.

B. Symbolic Execution Engine

We design a symbolic execution engine for DSL to generate test cases that can cover all execution paths described in the specification. The generated test cases are sent to the differential testing engine (§ V).

1) Workflow: We generate test cases for each path described in the DSL. To generate test cases for a path, we first determine each part of the instruction under test (i.e., the opcode and immediate arguments), and the operands taken by the instruction. We achieve this by solving its path constraint pc (i.e., the conjunction of constraints on the path) because an execution path will be invoked if its path constraint is satisfied. Then, to execute the generated test cases, we integrate each one into a pre-defined template along with the necessary instructions that set the operands.

Before exploring an execution path, we build symbols for the opcode and immediate arguments, as well as the operands which are taken by the instruction (§ IV-B2). Moreover, we define variables with which the instruction can interact.

We iterate over execution paths and generate test cases for each path. When a Statement contains a Constraint C, (i.e., an If Statement or an Assert Statement), and both C and its negation (denoted as $\neg C$) can be solved, we update their path constraints corresponding to whether C is satisfied. We construct the constraints (C and $\neg C$) indicated by the Condition according to the Condition type (§ IV-B4). For the path where C is satisfied, we update its path constraint to $pc \wedge C$, where pc denotes the path constraint of the original path. Similarly, for the other path, we update its path constraint to $pc \wedge \neg C$.

When the current execution path terminates (i.e., the symbolic execution ends at a Return Statement, a Trap Statement, an Assert Statement whose Condition is evaluated as false, or the last Statement of Statements), we generate test cases for this path. We achieve this by solving the path constraint pc to generate a set of value combinations, where each combination contains the opcode and immediate arguments that make up the instruction, and the operands taken by the instruction (§ IV-B5). Then, we wrap each combination as a test case (§ IV-B6). After generating test cases for the current path, we proceed to the next path and generate test cases for it. We repeat the above operations until all paths are explored.

2) Symbolization: We symbolize an immediate argument according to its type (e.g., $u32$), which is retrieved from the specification. Since an instruction takes operands from the stack, to symbolize operands, we model a symbolic stack, containing symbolized operands whose value and type are both symbols. We also symbolize the opcode so that we can obtain the candidates for the opcode that can satisfy a constraint determined by the opcode (i.e., a constraint constructed from $OpDefinedCond$).

3) Update symbols: There are two kinds of Statements that can update the symbols, i.e., Pop Statements and Let Statements. A Pop Statement "Pop($Alias$)" means to pop an operand from the stack and represent its value with an $Alias$ Variable. We take the Pop Statement "Pop($Alias(i)$)", which is transformed from the natural language "Pop a value $i32.const\ i$ from the stack", as an example to explain how we simulate a Pop Statement. To model its semantics, we first pop the operand on the symbolic stack top and assign its value to the symbol i, which is interpreted from $Alias(i)$.

Similarly, "Let(V1, V2)" represents assigning the evaluation result of "V2" to "V1". For example, a Statement "Let($Alias(ea)$, $Expr(i + memarg.offset)$)" represents using a Variable of name ea to represent the evaluation result of $i + memarg.offset$. To model the semantics of this Statement, we update the symbol ea to be $i+memarg.offset$.

4) Construct constraints: The basic idea that we generate constraints is to interpret the Condition in DSL. For each Condition and its negation, we construct a constraint according to its type. In the following paragraphs, we describe in detail how to generate constraints depending on the type of data being constrained [6].

Constraints on the types of operands. The satisfaction of these Conditions can be considered as constraints on the type of operands. We take the constraints constructed from $StackCond(N, T)$, which specifies the type of N operands on the symbolic stack top as T as an example. If the T is specified (e.g., $i32$) and the Condition is satisfied, we generate a constraint that requires the type of N operands on the stack top to be T. For its negation, we generate a constraint that the types of these operands are not T.

Constraints on the opcode. If an $OpDefinedCond(Op)$ is satisfied or not mentioned in the execution path, we construct a constraint $Op = Op_{defined}$, where Op represents the opcode of the instruction under test and $Op_{defined}$ is the opcode specified in the specification. Otherwise, there is a constraint $Op \in IllegalOps$, where $IllegalOps$ is a set of bytes that cannot be interpreted as an opcode.

Constraints on the value of operands and immediate arguments. The other constraints are decided by the value of operands and immediate arguments. We take the Condition $ExistCond(Elem(Instance, idx))$ as an example. $ExistCond(Elem(Instance, idx))$ is determined by whether an item in a variable $Instance[idx]$ exists, where idx denotes the index of the item and $Instance$ denotes a variable in the runtime, such as a linear memory. We interpret such a Condition as whether the idx is smaller than the length of $Instance$, which is defined in the template.

5) Infer symbol candidates: After a path is explored, we solve the path constraint to determine candidates of opcode, immediate arguments, operands (i.e., items in symbolic stack).

For constrained symbols, we infer their candidates by solving constraints. In addition, to cover more input space, for each symbol representing a value (i.e., the value of an immediate argument or an operand), we extend special values to the symbol's candidate set according to its type. TABLE I presents the special values for a subset of symbol types.

For an immediate argument or operand whose value is unconstrained, we generate value candidates according to its type. For an unconstrained immediate argument, we generate N random values as its candidates. Furthermore, we also extend the special values in TABLE I to the symbol's candidate set. For an operand whose value is unconstrained, we generate value candidates for all possible types of the operand.

After determining the candidates for symbols, we combine the opcode, immediate arguments, and operands. Each combination is wrapped as a test case (§ IV-B6).

TABLE I: Special values for partial types of data.

Value type	Special value set
$i32$	0, ± 1, $2^{31}-1$, -2^{31}
$i64$	0, ± 1, $2^{63}-1$, -2^{63}
$f32$	$\pm\infty$, ± 0, $-nan(0x7fc00000)$, $+nan(0xffc00000)$
$f64$	$\pm\infty$, ± 0, $-nan(0x7ff8000000000000)$, $+nan(0xfff8000000000000)$
$v128$	$byte(0x00)$x16, $byte(0xff)$x16

6) Generate test cases: We integrate each combination of opcode, operands, and immediate arguments as a test case. First, to execute the instruction I under test, we insert I, which consists of the opcode and immediate arguments, and a number of instructions to set the operands for I into the start function of the template introduced in § IV-B2. Then, we adjust the template to pass the validation. For example, to ensure the values left on the stack can be returned, we infer the type of the start function and then rewrite the function type by modifying the type section and function section in the template.

C. Mutator

To explore more input space and generate test cases of illegal encoding, we design a mutator, which mutates existing test cases to generate test cases as described in Algorithm 1.

We treat the test cases generated from the symbolic execution engine (§ IV-B) as seeds. Given a legal test case TC (§ IV-C1), we mutate its copy to get a new test case. Specifically, we achieve this by applying a randomly selected mutation strategy (§ IV-C2) on mutable bytes (§ IV-C3). We randomly generate N test cases by mutating TC. The mutated N test case will be sent to the differential testing engine (§ V) for testing. If it is legal, it will be treated as a seed that can

Algorithm 1: The Workflow of Mutator.

```
Input:
N_epoch: Mutation times limitation for one test case;
TCs: Test cases generated by the symbolic execution engine;
M: Mutation times limitation for one test case in TCs
forall TC ∈ TCs do
    Seeds ← {TC};
    mutationTimes ← 0;
    while Seeds ≠ φ do
        Seed ← Seeds.pop();
        isExecutable ← execute(TC);
        if isExecutable ∧ mutationTimes < M then
            forall i ∈ {1, 2, ..., N_epoch} do
                strategy ← randomMutation();
                possiblePosition ← getPosition(Seed);
                TC_M ← mutation(strategy, possiblePosition);
                Seeds ← Seeds ∪ {TC_M};
                mutationTimes ← mutationTimes + N_epoch;
            end
        end
    end
end
```

be mutated again. If the seed pool is empty or the number of test cases generated from TC up to M, the mutator will start to mutate the next legal test case generated by the symbolic execution engine.

1) Valid test case: We treat the test case that can be considered legal and instantiated by at least one runtime as a legal test case. We only mutate legal test cases to generate new test cases for the following reason. If a test case is treated as illegal by all runtimes, it indicates that the test case violates at least one validation rule on all runtimes. Because random mutation applied to this test case can hardly locate bytes causing the exception and fix them, the test cases generated by mutating an illegal test case can hardly pass the previously failed validation rule and will not invoke more code in the runtimes, wasting computing resources.

2) Mutation strategy: We design three mutation strategies. Each time we mutate a seed, we randomly choose a mutation strategy and apply it to mutable bytes.

1. Insert. Insert a random byte into the byte sequence.
2. Delete. Delete a random byte.
3. Replace. Choose a byte randomly and replace it with a byte with a random variable that obeys a uniform discrete distribution from 0 to 255.

Because a mutation may change the length of the function body (e.g., insert a byte into the function), we update the bytes representing the length of a function. we update the bytes representing the function size so that the mutated test case conforms to the rule that the actual function size should equal the declared size. Furthermore, to test whether the runtimes can reject a test case that violates this rule, we determine the new size to the actual length plus 1 or the actual length minus 1 with a probability of 1%, respectively.

3) Mutable bytes: The mutable bytes, which are the candidates for applying a mutation strategy are selective. First, we only mutate the bytes inside the start function (i.e., the function invoked by the runtime). Because we focus on the behavior of the start function, we do not mutate the bytes out of the start function, which cannot influence the semantics of the start function directly.

Second, we do not mutate the bytes corresponding to the instructions that initialize local variables or store local variables into global variables in the test cases, because these instructions are designed to check whether there exist differences in the local variables. If these instructions are broken by the mutation, the differential testing engine cannot dump the value of local variables (§ V-B).

V. Differential Testing Engine

We design a differential testing engine that can execute the test cases on the WebAssembly runtimes and detect the differences between these runtimes. To determine whether there is a difference between two runtimes, we model the state of a runtime with the variables in the runtime. After that, to identify differences among runtimes, we initialize the runtimes to the same state before execution and collect the state variables that describe the state of runtimes for comparison after the execution.

A. Data to Model the Runtime

We determine the variables that can model the state of the runtime. Therefore, we can initialize these variables in different runtimes to the same before the execution to guarantee the runtime have the same state before the instruction under test is executed. Similarly, we identify the difference among runtimes by comparing these variables. We use variables that are accessible to instructions and mutable to model a runtime. Since instruction can interact with the data in a function (i.e., the operand stack and local variables) and the global data (i.e., linear memory, global variables, and table), we consider all of them to model the state of a runtime. We denote the initial runtime state as S_I, which is represented as a tuple $\{L, Stack_{op}, Mem, G, T\}$. L and $Stack_{op}$ refer to the local variables and the operand stack in the start function (i.e., the function that will be tested). Mem, G, and T denote the linear memory, global variables, and table in a runtime, respectively.

We denote the runtime state after the execution by S_E. S_E is determined by the tuple $\{L, Stack_{op}, Mem, G, T, Exec\}$, where L, $Stack_{op}$, Mem, G, and T in S_E retain the same meanings as those in S_I. In addition, we use $Exec$ to denote the execution state (e.g., whether the execution is terminated).

Detecting WebAssembly runtime differences through differential testing can be formalized as follows. We use R to denote a WebAssembly runtime, tc to denote the input test case, and μ to represent a state variable in S_E. The process in which a runtime executes a test case can be represented as $S_E = R(tc)$. Given n runtimes under test, denoted as

$R_0, R_1, ..., R_{n-1}$, after the execution of a test case tc, we say there is a difference between them iff:

$$\exists i, j \in \{0, 1, ..., n-1\}, R_i(tc).\mu \neq R_j(tc).\mu$$

B. Our Strategy

We need to set each runtime under testing to the same initial state (i.e., S_I) and dump the state variables modeling S_E after the execution. Here, we introduce how we achieve this.

Since the runtimes are instantiated with the same input test case, they are expected to have the same S_I. We define the initial values of the state variables with two different pre-defined templates.

`Template-Empty`. We only define a start function in it, and no other variables are defined. We use this template to generate test cases for instructions that only interact with the stack.

`Template-Full`. We design this template to generate test cases for an instruction that interacts with variables in addition to the stack. For `Template-Full`, we initialize the state variables with the following strategy.

$\boldsymbol{Mem}$. Since comparing memory with a large size is time-consuming, we only define a linear memory with one page (64Kb) in `Template-Full`.

$\boldsymbol{G}$. In the `Template-Full`, we define global variables that cover all combinations of mutability (i.e., mutable, immutable) and number types [4] (i.e., $i32$, $f32$, $i64$, $f64$). Since some runtimes do not support the SIMD type or the reference type and they cannot parse a global variable of type $v128$ or reference, we do not define global variables of these two types in the template.

$\boldsymbol{L}$. In the `Template-Full`, we define four local variables of type $i32$, $f32$, $i64$, and $f64$, respectively. Similarly to the global variables, we do not define local variables of type $v128$ or reference.

$\boldsymbol{Stack_{op}}$. As introduced in § IV-B, we first determine the operands that will be taken by the instruction under test by the symbolic execution engine. Then, we insert instructions to push these operands on the stack preceding the instruction under test (§ IV-B6).

$\boldsymbol{T}$. We define a table containing function references in the `Template-Full`. To this end, we define three functions other than the start function, and the elements of the table are the function references that point to these three functions.

We dump and compare the state variables modeling S_E by different strategies after the execution of a test case.

$\boldsymbol{Mem}$ and $\boldsymbol{G}$. For each runtime, we dump the content of the linear memory $\boldsymbol{Mem}$ and global variables $\boldsymbol{G}$ by instrumenting the runtime under test.

$\boldsymbol{L}$. In some runtimes (e.g., wasmer), local variables may not be implemented as a concrete data structure [4]. In this case, we add instructions to move the value stored in local variables into pre-defined global variables. After that, we can retrieve the local variables from the dumped global variables.

$\boldsymbol{T}$. We only compare the size of the table rather than the content of the table. This is because the encoding of elements in a table (i.e., references) is opaque, and different implementations use different strategies to represent a reference.

$\boldsymbol{Stack_{op}}$. Since the value on the stack is expected to be returned as the function's return value, we instrument the runtime to collect the output of the start function, which should be the value on the stack. Since the encoding of a reference is opaque, if the value on the stack is a reference, we do not compare the value. To mitigate the non-determinism caused by an arithmetic NaN, if the value on the $Stack_{op}$ is an arithmetic NaN, we canonicalize it.

$\boldsymbol{Exec}$. We use $Exec$ to denote execution state, e.g., whether the execution times out, succeeds, or throws an exception.

VI. Evaluation

Implementation. We implement WADIFF with around 8,000 lines of Python in total. To solve the constraints generated from the symbolic execution, we use Claripy [7] as the SMT solver that is powered by Z3 [20].

Experiment Setup. All experiments were done on a server running Ubuntu 20.04 with an AMD Ryzen Threadripper 3960X 24-Core processor and 256G RAM.

Wasm Runtimes. We apply WADIFF to seven popular Wasm runtimes, i.e., wasmer [10], wasmi [11], WAMR-classic [15], WAMR-fast [16], wasm3 [8], WasmEdge [9], and WAVM [12]. Note that WAMR-classic and WAMR-fast refer to the WAMR in classic interpreter mode and fast interpreter mode, respectively.

Research Questions. In this section, we conduct experiments to evaluate WADIFF by answering the following questions.

- **RQ1**: Can the symbolic execution engine generate representative test cases?
- **RQ2**: Is the differential testing engine effective to detect differences?
- **RQ3**: What are the root causes of detected differences?

A. Specification Coverage of the Symbolic Execution Engine

We evaluate the effectiveness of the symbolic execution engine in terms of specification coverage, which is the number of covered paths determined by the value of operands and immediate arguments.

Approach. We construct a type-aware test case generator, which does not consider constraints on the value of operands and immediate arguments, as the baseline, to illustrate the effectiveness of the symbolic execution engine. Specifically, the type-aware generator knows the number of operands for each instruction and the correct type of each operand and immediate argument. When generating test cases for an instruction, it will generate operands and immediate arguments according to their types. Though such a generator can generate test cases covering all paths determined by constraints about the type of operands (e.g., Path 1 and Path 2 in Fig. 1(b)), it cannot cover the paths determined by the value of operands and immediate arguments sufficiently. Not all instructions have paths determined by values. For example, as shown in Fig. 1, the instruction *i32.add* only has the path determined by the opcode and type, but does not have the path determined by the value. There are 183 such paths in 82 instructions.

To demonstrate that WADIFF can generate more sufficient test cases than the type-aware generator, we compare the number of such paths that can be covered by the type-aware generator and WADIFF. For each instruction, we employ the type-aware generator to generate test cases in the same quantity as that generated by WADIFF.

Results. WADIFF can cover all the 183 paths. However, the type-aware generator can only cover 97 (53.01%) of them.

> **Answer to RQ1:** WADIFF can generate sufficient test cases, which can cover all the paths while a type-aware generator can only cover around half of them.

B. Differences Triggered by WADIFF

Approach. We conduct differential testing on the 7 runtimes. The symbolic execution engine takes 1,097 seconds to generate 296,662 test cases and 24,208 of them trigger differences. Furthermore, during the testing with a duration of 55,510 seconds (i.e., about 15 hours), the mutator generates 1,098,429 test cases, and 99,949 of them trigger differences. In total, 124,157 test cases trigger differences. We conduct the investigation on the test cases generated by the symbolic execution engine.

To explore the characteristics of the detected differences, we count the number of instructions affected by the differences in each instruction category defined in the specification.

Note that each runtime differs from at least one of the other runtimes when a difference is triggered. Therefore, to highlight the runtime that is more likely to exhibit unexpected behaviors, we say that there is a difference on a runtime R if the state of R differs from the states of half or more of the tested runtimes. When a difference is identified, all the runtimes will be analyzed for locating the root cause. Furthermore, we define two kinds of differences (i.e., execution difference and semantic difference) according to their features. We define **execution difference** to represent the difference where different runtimes have different execution states (e.g., time out, success). Given a runtime R whose execution state is different from half or more of the tested runtimes when executing a test case, we say there is an execution difference (D_E) on R. We also define **semantic difference** to represent the difference between the runtimes of the same execution state. Specifically, given N runtimes that can execute the test case, if the state of a runtime R is different from the state of other $N/2$ or more runtimes, it is a semantic difference (D_{Sem}) on R.

Results. Most runtimes can implement Variable instructions properly. However, we found that a number of instructions can trigger D_E on wasm3 because wasm3 fails to reject test cases where instruction takes operands of unexpected type as input. For the same reason, a number of instructions can trigger D_E on wasm3 when executing other instructions. For example, wasm3 can execute 23 Memory instructions that are rejected by most runtimes. Because some runtimes do not support SIMD instructions or Reference instructions, there are many instructions triggering D_E. For example, since wasmer, WAVM, and WasmEdge support SIMD instructions, test cases for 236 SIMD instructions can trigger D_E on these three runtimes. Because some SIMD instructions output a NaN value, whose encoding is not deterministic, there are 6 instructions that trigger D_{Sem} on these three runtimes. Most differences triggered by Table instructions are due to the different support for Table instructions. For example, wasm3 and wasmi do not support any Table instructions, while wasmer, WasmEdge, and WAVM support all of them. In the category of Numeric instructions, there are more instructions that can trigger D_{Sem}compared to other categories. We find that these differences are caused by the instructions (e.g., *f32.min*), which output a NaN value. D_E is triggered by 136 instructions on wasm3 because wasm3 executes instructions that take operands of unexpected type without exception, which is an unexpected behavior. Though there are only three Parametric instructions, these instructions trigger various differences. Because wasmer, WasmEdge, and WAVM support SIMD operands, test cases that contain SIMD operands can lead to D_E. Since wasmi has not implemented the instruction *select t* (i.e., the typed *select* instruction) and produces unexpected results when executing the instruction *select t*, a D_{Sem} is triggered on wasmi. Because wasmi and wasm3 do not support reference type, the test cases where an instruction (i.e., *select t* or *drop*) taking reference as input trigger D_E on these two runtimes. Test cases for Control instructions also trigger a difference on WasmEdge.

Because a single instruction may trigger D_{Sem} and D_E on the same runtime, the total count of instructions triggering D_{Sem} and D_E on a given runtime R may exceed the actual number of instructions triggering differences on R. For example, as shown in the last row and last column of the table, there are 16 instructions triggering D_{Sem}, but as all of them also trigger D_E, the total number of instructions triggering differences amounts to 417, rather than 433 (i.e. 417 + 16).

> **Answer to RQ2:** WADIFF can detect differences that affect 417 instructions. WADIFF generates 1,395,091 test cases and 124,157 of them trigger differences.

C. Root Cause of Differences

Approach. Analyzing the root causes of these differences is non-trivial because the number of test cases triggering differences is huge. To analyze the test cases more efficiently, we designed a semi-automatic method. Specifically, we categorize the differences according to the execution state and state variables in the runtime, and sample the test cases from each category for further investigation.

For differences where the runtimes have different execution states (e.g., times out), we collect the reason from the exception message (e.g., “SIMD is not supported”, “type mismatch”) and categorize them based on the reason.

For the differences in the runtimes that can execute test cases without exception, we use the modeled state variables (e.g., the stack, the global variables) in the runtime to categorize the differences. Furthermore, we utilize the attributes of

TABLE II: The number of instructions trigger differences. Each cell contains the number of instructions affected by differences, execution differences, and semantic differences. Note that one instruction may trigger several kinds of differences on the same runtime. Thus, the sum of instructions triggering D_{Sem} and D_E may be larger than the total number of instructions triggering differences on R.

	wasmer	wasmi	WAMR-classic	WAMR-fast	wasm3	WasmEdge	WAVM	Total
Variable instruction	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	2 / 2 / 0	0 / 0 / 0	0 / 0 / 0	2 / 2 / 0
Memory instruction	2 / 2 / 0	2 / 2 / 0	2 / 2 / 0	2 / 2 / 0	24 / 23 / 1	3 / 3 / 0	3 / 3 / 0	28 / 28 / 1
SIMD instruction	236 / 236 / 4	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	236 / 236 / 1	236 / 236 / 3	236 / 236 / 6
Reference instruction	1 / 1 / 0	3 / 3 / 0	0 / 0 / 0	0 / 0 / 0	3 / 3 / 0	1 / 1 / 0	1 / 1 / 0	3 / 3 / 0
Table instruction	0 / 0 / 0	8 / 8 / 0	2 / 2 / 0	2 / 2 / 0	8 / 8 / 0	1 / 1 / 0	1 / 1 / 0	8 / 8 / 0
Numeric instruction	4 / 0 / 4	2 / 0 / 2	6 / 0 / 6	2 / 0 / 2	136 / 136 / 4	8 / 0 / 8	2 / 0 / 2	136 / 136 / 8
Parametric instruction	3 / 3 / 0	2 / 2 / 1	0 / 0 / 0	0 / 0 / 0	3 / 3 / 0	3 / 3 / 0	3 / 3 / 0	3 / 3 / 1
Control instruction	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	1 / 1 / 0	0 / 0 / 0	1 / 1 / 0
Total	246 / 242 / 8	17 / 15 / 3	10 / 4 / 6	6 / 4 / 2	176 / 175 / 5	253 / 245 / 9	246 / 244 / 5	417 / 417 / 16

the stack value to distinguish the differences in the stack (e.g., whether the value is infinite) for a finer granularity.

In this case, the 124,157 test cases are divided into 286 categories. After our investigation, four root causes are located.

1) Different feature support: Since some runtimes do not support certain features, the runtimes have different behaviors on the instructions that require such features. For example, wasmi does not support SIMD instructions, while wasmer can support them. Therefore, wasmer can execute a SIMD instruction without exception, while wasmi throws an exception indicating that SIMD is not supported. Because such differences are foreseeable, a difference caused by different features supported by runtimes does not indicate a bug.

2) Different encoding rules: For specific floating values (e.g., NaN), the encoding rules are different among different runtimes. Though the specification recommends the encoding rule [1], [4], [35], runtimes still choose their preferred one, resulting in differences.

3) Different configurations: Some differences are due to the different configurations of the runtimes. For example, in WAMR, the maximum number of local variables that can be declared in a function is 65,536, while the number in wasm3 cannot exceed 32,768. Therefore, WAMR can execute a function with 40,000 local variables without exception, while wasm3 cannot, resulting in a difference. Because the specification does not specify the maximum number of local variables, such a difference does not indicate a bug.

4) Bugs: In total, we have detected 21 bugs, and 8 of them are confirmed or fixed (Table III). The found bugs are mainly due to the following root causes.

• **Incorrect decoding strategy**. Some runtimes may not implement the decoding phase precisely. In this case, test cases that violate the encoding rule may be decoded successfully, and vice versa. We found that four runtimes (i.e., WAMR-fast, WAMR-classic, wasm3, WasmEdge) successfully decoded the test cases with a wrong function size. Meanwhile, some runtimes may not reject the test cases containing an undefined opcode, i.e., an opcode not defined in the specification, resulting in 3 bugs disclosed.

• **Illegal alignment**. Alignment argument, which exists in store instructions (e.g., *i32.store*) or load instructions (e.g., *i32.load*), should be an integer power of 2 (e.g., 1, 2, 4,...) and not be larger than natural. However, we find that four runtimes (i.e., wasmi, WAVM, wasmer, wasm3) fail to reject test cases with an illegal alignment argument.

• **Wrong instruction implementation**. We find 4 bugs caused by the incorrect instruction implementation. For example, WAVM, WAMR-classic, and WAMR-fast fail to throw an exception indicating out-of-bounds for an *table.init* instruction, which violates the requirements on the specification. In WAMR-classic and WAMR-fast, an instruction *ref.func x* in the function with index x can unexpectedly result in an exception, even though it is a legal instruction.

• **Others**. We also detected 6 bugs whose root causes are various. For example, we found that a test case where an instruction takes operands of unexpected type as input may pass the validation of wasm3 and WAVM, which is an unexpected behavior.

• **Bug Impact Analysis**. There are 5 bugs lying in the execution phase, and they are triggered by legal test cases (i.e., the bugs #3, #4, #6, #7, #13 in Table III). These bugs make the runtime have unexpected execution behaviors. For example, bug #3 in WAMR-classic and WAMR-fast leads to an unexpected termination of execution when executing a legal *ref.func x* instruction. Bug #7 in wasmi causes the runtime to produce incorrect output for the instruction *select t*. This can result in unexpected execution results, which may violate the original semantics. The impact is rather large, especially in blockchain ecosystem as the contract's semantics may be totally changed, resulting in unexpected loss.

Illegal test cases trigger the other 16 bugs, which lie in the decoding and validation of runtimes. These bugs make the runtime fail to reject an illegal test case.

Answer to RQ3: We found 4 root causes of differences. A difference may be caused by different feature support, encoding rules, configurations and bugs.

VII. Threat to Validity

A. Unsupported Instructions

WADIFF can generate test cases for 428 instructions out of 437 instructions described in the specification. WADIFF cannot

TABLE III: Detail description of the detected bugs.

#	Runtime	Root cause	State	Link
1	WAMR-fast & WAMR-classic	Incorrect decoding strategy	Fixed	https://github.com/bytecodealliance/wasm-micro-runtime/issues/1463
2	WAMR-fast & WAMR-classic	Incorrect decoding strategy	Fixed	https://github.com/bytecodealliance/wasm-micro-runtime/issues/1474
3	WAMR-fast & WAMR-classic	Wrong instruction implementation	Fixed	https://github.com/bytecodealliance/wasm-micro-runtime/issues/2093
4	WAMR-fast & WAMR-classic	Wrong instruction implementation	Fixed	https://github.com/bytecodealliance/wasm-micro-runtime/issues/2096
5	WasmEdge	Incorrect decoding strategy	Fixed	https://github.com/WasmEdge/WasmEdge/issues/2080
6	WasmEdge	Others	Confirmed	https://github.com/WasmEdge/WasmEdge/issues/2079
7	wasmi	Wrong instruction implementation	Fixed	https://github.com/paritytech/wasmi/commit/c15d525adc8358582d4c65fa25d0ca9f49655625
8	wasmi	Illegal alignment	Confirmed	https://github.com/paritytech/wasmi/issues/570
9	wasmer	Illegal alignment	Reported	https://github.com/wasmerio/wasmer/issues/3846
10	WAVM	Illegal alignment	Reported	https://github.com/WAVM/WAVM/issues/346
11	WAVM	Others	Reported	https://github.com/WAVM/WAVM/issues/354
12	WAVM	Incorrect decoding strategy	Reported	https://github.com/WAVM/WAVM/issues/357
13	WAVM	Wrong instruction implementation	Reported	https://github.com/WAVM/WAVM/issues/360
14	wasm3	Others	Reported	https://github.com/wasm3/wasm3/issues/399
15	wasm3	Illegal alignment	Reported	https://github.com/wasm3/wasm3/issues/407
16	wasm3	Others	Reported	https://github.com/wasm3/wasm3/issues/426
17	wasm3	Incorrect decoding strategy	Reported	https://github.com/wasm3/wasm3/issues/428
18	wasm3	Incorrect decoding strategy	Reported	https://github.com/wasm3/wasm3/issues/429
19	wasm3	Incorrect decoding strategy	Reported	https://github.com/wasm3/wasm3/issues/431
20	wasm3	Others	Reported	https://github.com/wasm3/wasm3/issues/430
21	wasm3	Others	Reported	https://github.com/wasm3/wasm3/issues/427

generate test cases for 9 control instructions (e.g., *if*, *br*), because the natural language describing the behaviors of these instructions is more complex, which cannot be handled by the current version of the DSL transformer. To generate test cases for these instructions, we need to design new patterns and DSL grammar to represent the semantics of these instructions. In addition, we need to update the symbolic execution engine to interpret the newly introduced DSL grammar. We will utilize more powerful NLP techniques to design a DSL transformer that can handle these instructions and generate test cases for them in the future.

B. Comparison of References

We skip the comparison of references (i.e., a type of data in WebAssembly) in the table and stack because the specification does not specify the rule for encoding a reference value. In addition, we observe that different runtimes use different strategies to encode a reference. This strategy makes us cannot detect the differences in the references in the table or the stack. It is non-trivial to retrieve the reference from the encoding in different runtimes because it requires a lot of manual work to investigate how each runtime encodes a reference. In the future, we will design an encoding reversing engineering approach to enable WADIFF to retrieve and compare references.

VIII. Discussion and Future Work

The current generated test cases usually consist of one target instruction. However, the bugs that can only be triggered by a sequence of instructions may not be explored. To overcome this limitation, we will enhance WADIFF by generating sequences of instructions. Meanwhile, the DSL and the symbolic execution engine in WADIFF are now used for generating test cases only. We will further apply DSL and symbolic execution engine on generating function-level test cases.

WADIFF now does not have very high code coverage on the tested runtimes. For example, while the test suite of WAMR-fast achieves a code coverage (line coverage) of 43.5% on WAMR-fast, the test cases generated by the symbolic execution engine of WADIFF only achieve a coverage of 31.4%. The coverage of WADIFF is limited for two reasons. First, there are 9 instructions that are currently not supported by WADIFF (§ VII-A). The code to implement these instructions cannot be covered by the current version of WADIFF. Second, the focus of WADIFF is on testing the behavior of instructions rather than the instantiation of modules. As a result, the code for instantiating a module is not explored in depth. Currently, we have designed two templates that define two modules for test case generation. However, since these templates do not incorporate complex grammar, the test cases generated by WADIFF cannot explore the code for instantiating a module in depth. Note that, though the code coverage of WADIFF is lower than that of the WAMR-fast test suite, WADIFF can detect bugs that cannot be detected by the WAMR-fast test suite. To improve code coverage, we plan to enable WADIFF to generate test cases for all WebAssembly instructions and create test cases involving various modules.

The current differential testing engine skips the comparison of the references in the stack and table due to the diverse encoding rules across different runtimes. To address this, we will investigate the encoding rules for references in different runtimes so that we can retrieve and compare the references in the future.

IX. Related work

WebAssembly Security. There are many studies on WebAssembly security [32], [22], [28], [26], [19]. Zhang et al. [36] conducted research on bugs in WebAssembly runtimes and proposed a bug detector based on the patterns of bugs they detected. However, because the detector is pattern based, it cannot detect bugs with an unseen pattern. Daniel et al. [27] analyzed the utilization of classic attack primitives in WebAssembly programs. However, their research focuses on the vulnerabilities in WebAssembly programs and compilers, rather than the bugs in WebAssembly runtimes. Jiang et al. proposed the fuzzing framework WasmFuzzer [24] to detect the bugs in WebAssembly runtimes. However, it contains two limitations. First, it cannot detect non-crash bug because it only uses the system signals to detect bugs. Second, the test cases it generates are less representative than the test cases

generated by WADIFF, because it generates test cases by assigning random operands to instructions.

Differential Testing. Differential testing has been employed to detect bugs in implementations following the same specification [21], [33], [25], [21]. Martignoni et al. [31] proposed a differential testing framework named PokeEMU, which generates test cases by using symbolic execution on a faithful emulator to detect bugs in x86 CPU emulators. However, it cannot be guaranteed that the chosen emulator follows the specification strictly and there is no bug in it. Jiang et al. proposed a differential testing framework Examiner [25], which implements a symbolic execution engine for ARM architecture specification language (ASL) to generate test cases. However, this strategy cannot be applied to testing WebAssembly runtime directly, because the semantics of each instruction is not described in ASL. Furthermore, in addition to using the symbolic execution engine to generate test cases according to the semantics of each instruction, WADIFF employs a mutator to generate test cases with illegal encoding. Hwang et al. proposed JUSTGen [23] to generate test cases for testing JNI functions. It uses a symbolic execution engine to generate test cases according to the specification of JNI in DSL. However, this approach cannot be employed to test WebAssembly instructions directly as WebAssembly runtimes and JVMs have different runtime states.

There exists research [21] that uses differential testing to detect the bugs in WebAssembly runtimes. However, they cannot generate representative test cases, which is randomly generated, and whether all behaviors of instruction are covered is unknown. Furthermore, when detecting differences, it does not compare the local variables and the table. If there is a difference between these two kinds of attributes, the framework cannot detect it. WADIFF fills this gap and can detect various bugs.

X. CONCLUSION

We design a differential testing framework named WADIFF for WebAssembly runtimes. Our evaluation shows that WADIFF can generate representative test cases. By conducting differential testing, we found 417 inconsistent instructions caused by the different implementations of runtimes (e.g., different features supported and resource configurations) and bugs in runtimes. Furthermore, we identified 21 bugs and 8 of them are confirmed or fixed.

XI. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments. This research is partially supported by Hong Kong RGC Project (No. PolyU15224121) and HK ITF Project (No. PRP/005/23FX).

REFERENCES

[1] "Wasm design rationale," https://github.com/WebAssembly/design/blob/main/Rationale.md, 2020.

[2] "The main page of eosio," https://eos.io/, 2022.

[3] "The main page of near," https://near.org/, 2022.

[4] "The main page of webassembly.org," https://webassembly.org/, 2022.

[5] "Conditions," https://github.com/erxiaozhou/WaDiff/tree/main/material/Conditions.pdf, 2023.

[6] "Constraintstable," https://github.com/erxiaozhou/WaDiff/tree/main/material/ConstraintsTable.pdf, 2023.

[7] "The main page of claripy," https://docs.angr.io/en/latest/advanced-topics/claripy.html, 2023.

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

[9] "The main page of wasmedge," https://wasmedge.org/, 2023.

[10] "The main page of wasmer," https://github.com/wasmerio/wasmer/, 2023.

[11] "The main page of wasmi," https://github.com/paritytech/wasmi, 2023.

[12] "The main page of wavm," https://github.com/WAVM/WAVM, 2023.

[13] "The main page of webassembly micro runtime," https://bytecodealliance.github.io/wamr.dev/, 2023.

[14] "Variables," https://github.com/erxiaozhou/WaDiff/blob/main/material/Variables.pdf, 2023.

[15] "Wamr classic interpreter," https://github.com/bytecodealliance/wasm-micro-runtime/blob/main/core/iwasm/interpreter/wasm_interp_classic.c, 2023.

[16] "Wamr fast interpreter," https://github.com/bytecodealliance/wasm-micro-runtime/blob/main/core/iwasm/interpreter/wasm_interp_fast.c, 2023.

[17] "Wasmtime cves," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=WebAssembly, 2023.

[18] J. C. Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-mutate: Fuzzing webassembly compilers with e-graphs," 2022.

[19] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 703–715.

[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

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

[22] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, 2021, pp. 2696–2708.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 217–234.

[28] D. Lehmann and M. Pradel, "Finding the dwarf: Recovering pecise types from webassembly binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 410–425.

[29] B. Li, W. Dong, and Y. Gao, "Wiprog: A webassembly-based approach to integrated iot programming," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[30] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, pp. 261–272.

[31] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 337–348, 2012.

[32] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.

[33] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*, vol. 98. Citeseer, 1998, pp. 618–622.

[34] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 2031–2042.

[35] C. Watt, "Mechanising and verifying the webassembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, 2018, pp. 53–65.

[36] 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.