RESEARCH-ARTICLE

WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation

Published: 11 September 2024

Citation in BibTeX format

ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis
September 16 - 20, 2024
Vienna, Austria

Conference Sponsors:
SIGSOFT

SHANGTONG CAO, Beijing University of Posts and Telecommunications, Beijing, Beijing, China

NINGYU HE, Peking University, Beijing, China

XINYU SHE, Huazhong University of Science and Technology, Wuhan, Hubei, China

YIXUAN ZHANG, Peking University, Beijing, China

MU ZHANG, The University of Utah, Salt Lake City, UT, United States

HAOYU WANG, Huazhong University of Science and Technology, Wuhan, Hubei, China

Open Access Support provided by:

Peking University

Huazhong University of Science and Technology

Beijing University of Posts and Telecommunications

The University of Utah

WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation

Shangtong Cao
Beijing University of Posts and Telecommunications
Beijing, China
shangtongcao@bupt.edu.cn

Ningyu He
Peking University
Beijing, China
ningyu.he@pku.edu.cn

Xinyu She
Huazhong University of Science and Technology
Wuhan, China
xinyushe@hust.edu.cn

Yixuan Zhang
Peking University
Beijing, China
zhangyixuan.6290@pku.edu.cn

Mu Zhang
University of Utah
Salt Lake City, USA
muzhang@cs.utah.edu

Haoyu Wang*
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

A fundamental component of the Wasm ecosystem is the Wasm runtime, as it directly impacts whether Wasm applications can be executed as expected. Bugs in Wasm runtimes are frequently reported, so the research community has made a few attempts to design automated testing frameworks to detect bugs in Wasm runtimes. However, existing testing frameworks are limited by the quality of test cases, i.e., they face challenges in generating Wasm binaries that are both semantically rich and syntactically correct. As a result, complicated bugs cannot be triggered effectively. In this work, we present WASMaker, a novel differential testing framework that can generate complicated Wasm test cases by disassembling and assembling real-world Wasm binaries, which can trigger hidden inconsistencies among Wasm runtimes. To further pinpoint the root causes of unexpected behaviors, we design a runtime-agnostic root cause location method to locate bugs accurately. Extensive evaluation suggests that WASMaker outperforms state-of-the-art techniques in terms of both efficiency and effectiveness. We have uncovered 33 unique bugs in popular Wasm runtimes, among which 25 have been confirmed.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

WebAssembly, WebAssembly Runtime, Binary Generation, Differential Testing

*Corresponding author

ACM Reference Format:
Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2024. 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 (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680358

1 Introduction

WebAssembly (Wasm), a low-level bytecode format, was proposed by several Internet giants in 2017 [14]. Due to its excellent portability, native-like speed, compact size, and safety guarantee, Wasm is gaining growing popularity. By 2022, more than 97% of existing browsers have supported Wasm. Beyond web browsers, Wasm has been favored in a wide range of domains, including mobile apps, blockchain, IoT, etc. Wasm can be regarded as the compilation target for almost all mainstream high-level programming languages, e.g., C, C++, Go, and Rust [26, 32, 45]. Wasm binaries are executed in Wasm runtimes, which are similar to the virtual machine that serves as an intermediate layer between the Wasm binaries and the underlying system. Currently, lots of Wasm runtimes have been implemented and actively maintained on GitHub, like Wasmtime [38], Wasmer [37], and WasmEdge [36].

Wasm runtimes play a key role in the ecosystem, as it directly impacts whether Wasm applications can be executed as expected. However, a variety of Wasm runtime-specific bugs have been reported from time to time. For example, Zhang et al. [44] have empirically analyzed over 300 real-world bugs of Wasm runtimes and created a taxonomy of 31 bug categories of Wasm runtimes. Thus, some fellow researchers in our community proposed to develop automated methods for detecting bugs in Wasm runtimes. For example, Jiang et al. [18] have uncovered that Wasm runtimes may not correctly execute Wasm binary by adopting coverage-guided fuzzing. WADIFF [46] further adopted symbolic execution to generate lots of Wasm binaries and conducted differential testing to identify implementation bugs of runtimes.

Although recent automated testing approaches have shown promising results in identifying Wasm runtime bugs via Wasm binary generation, they are limited by their inability to generate semantically rich binaries (see §2.2). Thus, complicated bugs cannot be triggered. For example, WADIFF can only generate small Wasm binaries that target a single instruction, while real-world Wasm binaries are indeed much more complex. Specifically, there are over 430 kinds of instructions and 13 types of sections with different functionalities in Wasm, indicating that Wasm is a bytecode format with rich semantics. For comprehensive testing, ensuring that the generated Wasm binaries can cover as many semantics as possible is crucial and necessary. Further, it is impossible to generate Wasm binaries arbitrarily, as each Wasm binary should be validated for syntactic correctness before executing.

This work. We present WASMaker, a novel differential testing framework for Wasm runtimes that can generate syntactically correct and semantically rich Wasm binaries. We have designed a dedicated algorithm to extract basic elements from real-world Wasm binaries and randomly assembled them into Wasm binaries with valid syntax and rich semantics. WASMaker further applies different levels of mutation strategies (e.g., AST-level and module-level mutation) on generated binaries to increase their diversity, and sends them to Wasm runtimes simultaneously to investigate Inconsistencies. Because different binaries may lead to the same inconsistent state, to further pinpoint the root causes of inconsistencies, we proposed a root cause identification algorithm that can accurately locate bugs at the function and instruction level. Extensive experiments show the superiority of WASMaker over state-of-the-art techniques in both terms of efficiency and effectiveness. WASMaker can generate 6.0x and 148.8x Wasm binaries that can lead to inconsistencies over two baselines. By applying WASMaker on four representative Wasm runtimes, over 167K Wasm binaries that can lead to inconsistencies are generated, attributed to 33 unique bugs. With our timely disclosure, 25 bugs have been confirmed by runtime developers, and 11 have been fixed with our aid by the time of this writing.

The main contributions of this work are as follows:

- We propose WASMaker, a novel differential testing framework that can generate syntactically correct and semantically rich Wasm binaries by disassembling and assembling real-world Wasm binaries, which significantly increases the diversity of generated Wasm binaries in terms of semantics.
- We design a runtime-agnostic root cause identification algorithm that can accurately pinpoint the location of bugs, which significantly eases the burden of runtime developers for further verification and bug patching.
- WASMaker has identified 33 unique bugs that can lead to unexpected behaviors for mainstream Wasm runtimes, among which 25 have been confirmed and 11 have been patched with our timely disclosure.

2 Background & Motivation

2.1 WebAssembly & Runtime

WebAssembly (Wasm) is an emerging stack-based binary format that can be compiled from mainstream high-level languages. Except for the original four primary data types, i.e., `i32`, `i64`, `f32`, and `f64`, `v128` has been recently introduced to support SIMD instructions [41]. Each Wasm binary is composed of 13 sections [43], and complex functionalities can only be achieved by coupling sections. For example, implementing a function involves three sections: the *type section* for function signatures, the *code section* for local variables and the function body, and the *function section* for mapping type indexes to function indexes. Each Wasm binary will be statically validated on its syntactic validity before executing. The validation mainly focuses on the stack. It checks if the operands match the specified types and ensures stack balance, verifying that each block and function behaves according to its signature.

Wasm runtimes provide an executing environment for Wasm binaries in various hardware and operating systems [44]. They play a vital role in supporting Wasm-based functionalities in blockchain platforms [12] and embedded devices [33], enabling the deployment of lightweight, high-performance applications in resource-sensitive environments. Additionally, except for the inefficient *interpreting* mode, both *JIT (Just-In-Time)* and *AOT (Ahead-Of-Time)* compilations are adopted by some runtimes to improve the performance. Wasm runtime is also responsible for handling interactions between Wasm binaries and the external environment. In the early stage, each runtime has its specific set of compiling toolchains and wrappers for APIs exposed by the operating systems, which results in a severe compatibility issue. Thus, WebAssembly System Interface (WASI) [39] emerges, which defines the function signatures of each API as well as its behavior. Currently, WASI is supported by lots of mainstream Wasm runtimes as well as their compiling toolchains.

2.2 Motivation

Improper implementation of Wasm runtimes will significantly hamper the intended design goal of Wasm, i.e., *security* and *efficiency*. However, testing the correctness of the implementation of Wasm runtimes is challenging. Currently, only differential testing, one of the dynamic analysis methods, is adopted by existing studies [44, 46]. This is because the static validation may struggle with the complex logic in runtimes, and the pre-defined rules very likely import false positives. Differential testing is a widely adopted technique that compares the outputs or states among different targets (i.e., different implementations of the same functionality) while giving an identical input. Moreover, it is independent of oracles, one of the main challenges faced by other dynamic analysis methods, like grey-box fuzzing. Although differential testing seems to be a promising approach, there still exist some challenges for detecting bugs in Wasm runtimes, which can be summarized as follows:

Challenge #1: Generating syntactic-correct and semantic-rich Wasm binaries. As we mentioned in §2.1, each Wasm binary should be validated for syntactic correctness before executing. Further, in Wasm, there exist over 430 instructions and 13 types of sections with different functionalities, indicating that Wasm is a bytecode format with rich semantics. To reach the goal of comprehensive testing, instead of guaranteeing syntactic correctness, it is also crucial and necessary to ensure that the generated Wasm binaries can cover as many semantics as possible. In other words, *the generated Wasm binaries should be syntactic-correct and semantic-rich.* As for syntactic correctness, the generated binaries should be stack-balanced, and the index reference among sections should be correct. As for the semantic richness, on the one hand, the instructions in the generated Wasm binary should interact with as many sections as possible. On the other hand, these instructions should be covered as much as possible during execution at runtime.

```
1  (func $Wasm-smith (type 1)
2    (param i32) (result i32 f32)
3    block
4      local.get 0
5      ...
6      block
7        br 1
8        ...
9      end            Jump
10     ...
11   end
12   i32.const 60812
13   f32.const 1.89)
```

Figure 1: A Wasm binary from wasm-smith.

```
(module
  (type (;0;) (func (result i32)))
  (func $WADIFF (type 0)
    (local i32 i32)
    ...
    i32.const 5
    i32.const 10
    i32.add  ;; target instruction
    ...)
  (memory (;0;) 1)
  (export "memory" (memory 0))
  (export "_start" (func $WADIFF))
  (data (;0;) (i32.const 0) "mem"))
```

Figure 2: A Wasm binary from WADIFF.

Challenge #2: Error localization to refine Wasm binaries according to root causes. During differential testing, pinpointing the root cause of inconsistencies among different runtimes is challenging. Differential testing generates numerous Wasm binaries, each with hundreds or thousands of instructions, making it difficult to identify the specific function or instruction causing a bug. However, implementing the process of bug diagnosing is challenging because not all Wasm runtimes come with developed debugging tools. Moreover, handling the compatibility issue among these runtimes also raises the concern of scalability.

Limitations of current tools. To the best of our knowledge, only two tools are available for differential testing Wasm runtimes, i.e., wasm-smith [3] and WADIFF [46]. Specifically, wasm-smith is a Wasm test case generator proposed by the official community. It randomly selects instructions while considering the stack balance to guarantee syntactic correctness. As for WADIFF, it adopts symbolic execution to generate test cases for each instruction according to the specification. However, both tools have certain limitations.

Specifically, wasm-smith ignores the semantics of the generated Wasm binaries, hindered by the **Challenge #1**. As shown in Figure 1, the `br` instruction at L7 will direct the control flow to L11, i.e., the end of the function. In other words, instructions between L7 and L11 will be ignored by runtimes. Additionally, L2 indicates that the results should be an `i32` and an `f32`. To ensure the stack balance, wasm-smith simply pushes two constant values (L12 and L13) of the corresponding types. This means that this function can be aggressively optimized to only include the last two instructions. Consequently, wasm-smith can only generate Wasm binaries composed of lots of meaningless instructions, which hampers both the efficiency and effectiveness of differential testing.

As for WADIFF, which adopts symbolic execution on the Wasm specification of each instruction to generate test cases, it can only generate simple Wasm binaries (~10 – 100 instructions) to verify runtimes. As shown in Figure 2, this Wasm binary verifies the implementation of `i32.add`. WADIFF follows the Occam's Razor principle [1], i.e., following the simplest control flow and giving only the necessary parameters generated by constraints. In other words, it is hard for WADIFF to cover complex functionalities among 13 sections within a single Wasm binary. Moreover, it can only generate a finite number of Wasm binaries after traversing all possible paths for each instruction, and its random mutation method is likely to generate invalid ones [46]. Last but not least, generating test cases against control instructions is not yet supported by WADIFF, e.g., `call_indirect` and `loop`. In summary, WADIFF also faces the **Challenge #1** we proposed previously.

Our approach. To address these two challenges, we come up with some key ideas. Specifically, to generate syntactic-correct and semantic-rich Wasm binaries, we extract AST nodes from existing real-world Wasm binaries and randomly assemble them in a syntactic-correct way. Moreover, to increase the diversity of semantics, we also import some mutations on the AST level and module level, like introducing SIMD instructions. As for the error localization issue, we implement a static instrumentation-based error localization algorithm, which is runtime-agnostic to perform the function level or even instruction level of root cause localization.

3 Approach

3.1 Overview

The workflow of WASMaker is depicted in Fig 3, which can be divided into three phases, i.e., *corpus preparation*, *binary generation & mutation*, and *differential testing*. Specifically, in the corpus preparation phase, based on all collected real-world Wasm binaries, WASMaker first parses ASTs and extracts valid AST sub-trees from them. Then, in the binary generation & mutation phase, WASMaker randomly assembles AST sub-trees as a valid Wasm binary. To enhance the diversity of the generated Wasm binaries, WASMaker performs AST-level and module-level mutations. Last, in the differential testing phase, WASMaker sends a Wasm binary to multiple Wasm runtimes simultaneously to investigate *inconsistent behaviors*. Taking advantage of static instrumentation, WASMaker can conduct a runtime-agnostic error localization to the root cause in Wasm binaries that lead to such inconsistent behaviors. We detail these three phases in the following.

3.2 Corpus Preparation

Instead of directly generating Wasm binaries, we decide to take real-world Wasm binaries that possess rich semantics as basic elements to assemble Wasm binaries. To be specific, we first extract their abstract syntax trees (ASTs), and split them into sub-trees. Then, we sample AST sub-trees from all extracted ones, and assemble them in a syntactic-valid way. Thus, the corpus preparation phase can be divided into steps including *context extraction*, *AST parsing*, and *post-processing*, which are depicted in Algorithm 1.

3.2.1 *Context Extraction.*

Wasm is a statically typed language, and each Wasm binary will be comprehensively validated syntactically and semantically before being executed. Specifically, *on the syntactic side*, Wasm is a stack-based language. Therefore, each instruction will consume or push a certain number of operands from or onto the stack. For example, `i32.load` requires an `i32` operand as the address and pushes the retrieved `i32` data onto the stack. For some instructions, the number of required arguments is variable, like `call` and `block`. Thus, guaranteeing the stack balance needs to consider the signature of the callee and all included instructions for these two instructions, respectively. *On the semantic side*, a Wasm binary is composed of 13 sections with different functionalities. Without considering the semantic validity, a stack-balanced Wasm binary still cannot pass the validation. For example, if we have assembled a function `foo`, which invokes `bar`, we should guarantee the existence of `bar`: not only its implementation in the *code section*, but also its function signature declared in the *type section* and the mapping relation kept in the *function section*.

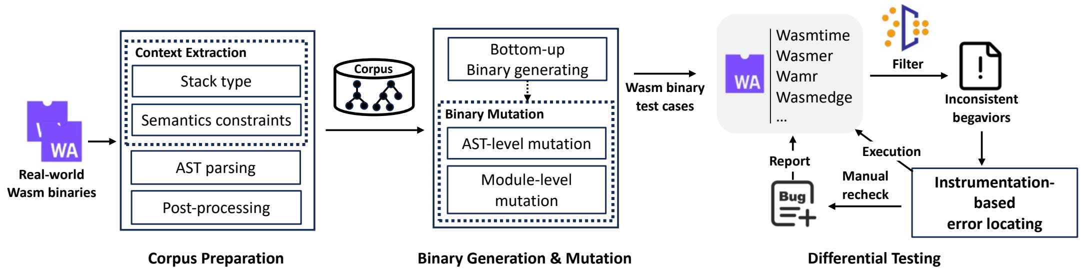


Figure 3: The architecture and workflow of WASMaker.

Table 1: Instruction types and the representations, as well as the stack type and semantic constraint should be guaranteed.

Instruction	Stack Type	Semantic Constraint
Numeric Instructions		
i32.const c	$[] \to [i32]$	
i64.add	$[i64, i64] \to [i64]$	
Vector Instructions		
v128.const c	$[] \to [v128]$	
i32x4.add	$[v128, v128] \to [v128]$	
Parametric Instructions		
select	$[t, t, i32] \to [t]$	
drop	$[t] \to []$	
Variable Instructions		
local.get n	$[] \to [t]$	`Local` v `idx`$_v = n$ `type`$_v = t$
global.set n	$[t] \to []$	`Global` v `idx`$_v = n$ `type`$_v = t$
Memory Instructions		
i32.load *memarg*	$[i32] \to [i32]$	`Memory` m `Limit` min, max `page`$_m \in [min, max]$
Table Instructions		
table.get n	$[i32] \to [t]$	`Table` tb `idx`$_{tb} = n$
Control Instructions		
call n	$[t^*] \to [t^*]$	`Function` f `Signature` $s = [t^*] \to [t^*]$ `signature`$_f = s$ `idx`$_f = n$
call_indirect n	$[t^*, i32] \to [t^*]$	`IndirectFunction` f `Signature` $s = [t^*] \to [t^*]$ `signature`$_f = s$ `idx`$_s = n$
block n	$[t^*] \to [t^*]$	`Signature` $s = [t^*] \to [t^*]$ `idx`$_s = n$

To solve this challenge, we first categorize Wasm instructions into seven groups according to their *context*. Specifically, the context of an instruction consists of its *stack type* and *semantic constraints* (the must-satisfy conditions to guarantee semantic validity). Table 1 illustrates the representatives of each group. Specifically, the stack type of numeric instructions is fixed, and there are no extra semantic constraints on them. Vector instructions are similar to them but are specifically designed for the vector type (see §2.1). In addition, the stack type of parametric instructions may vary. For example, `drop` consumes an operand, regardless of its type. Thus, the stack type of `drop` is $[t] \to []$, where t is a wildcard and refers to *any type*. The other four types of instructions need semantic constraints. For example, `local.get` n has two semantic constraints: for a local variable v, its index and type should be n and t, respectively. As for memory and table instructions, they are respectively responsible for interacting with the memory and table area. For instance, `i32.load` requires the number of pages of the linear memory to be within a certain range, i.e., $[min, max]$, as declared in the binary. Table instructions require that the accessed table exists. In Wasm, there are two kinds of function invocations: `call` and `call_indirect`, which have subtle differences in semantic constraints. For `call` n, it directly invokes the n-th function, while the n of `call_indirect` indicates the index of the function type declared in the type section. The index of the callee is retrieved dynamically from the stack at runtime, i.e., the second `i32` parameter of its stack type.

Algorithm 1 The algorithm of corpus preparation.

```
Input: instrs - list of instructions, binary - the Wasm binary
Output ASTs - the corpus, composed of a list of AST
1: function CorpusPreparation(instrs, binary)
2:     ASTs ← InitializeList()
3:     for each instr ∈ instrs do
4:         context ← GetContext(instr, binary)                    ▷ §3.2.1
5:         node ← Node(instr, context)
6:         if instr.opcode ∈ [block, loop, if] then               ▷ §3.2.2
7:             node.child ← CorpusPreparation(instr.args, binary)
8:         end if
9:         params ← context.type.params
10:        while params do
11:            for each preNode ∈ Reversed(ASTs) do
12:                node ← AppendChild(node, preNode)
13:                ASTs.pop()
14:                params.pop()
15:            end for
16:        end while
17:        ASTs.append(node)
18:    end for
19:    ASTs ← PostProcessing(ASTs)                                ▷ §3.2.3
20:    return ASTs
21: end function
```

To build the corpus, the algorithm first goes through the instruction list (L3) and extracts context information for each of them to build the node corresponding to each instruction (L4 – L5). Specifically, against each instruction, the algorithm extracts its stack type, which can be analyzed statically (see §2.1). As for the semantic constraints, it is parsed by indexing all necessary sections within the given binary. Both of them will be packed within a variable, named *context*, which is then treated as an attribute of a *Node* instance along with the instruction (*instr*). Take the `call` instruction as an example, where Listing 1 illustrates its concrete context extraction process. To be specific, the algorithm first initializes a fresh `Function` object (L3). Then, it extracts the immediate number n of `call`, and obtains its function signature s by indexing through the function and type sections (L5 – L7). According to its semantic constraint, the callee is linked to s (L9). Last, both the stack type and semantic constraint are packed and returned (L10).

```
 1 # suppose the instr and binary are given
 2 # initialize a Function
 3 Function f = Function()
 4 # get the callee's signature
 5 n = instr.args
 6 typeid = binary.functionSec[n]
 7 Signature s = binary.typeSec[typeid]
 8 # build the context
 9 Semantics sem = Semantics(function=f, signature=s)
10 return {'type': s, 'semantics': sem}
```

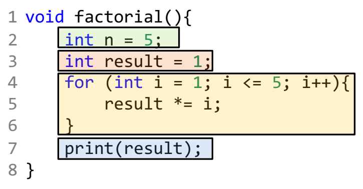
Listing 1: The pseudocode of `GetContext` on `call`.

3.2.2 *AST Parsing.*

Parsing ASTs of real-world Wasm binaries is necessary before extracting sub-trees from them to build the corpus. By leveraging the extracted context mentioned in §3.2.1, a Wasm binary can be easily parsed to the corresponding ASTs. To better illustrate how this process happens, Fig. 4a to Fig. 4c shows a factorial function written in C, the instruction list of the compiled Wasm binary, and the corresponding AST, respectively. We detail the process combining with Algorithm 1. As we can see from Fig. 4b, the compiled instruction list is quite flat, which makes it hard to identify the AST structure. Therefore, the algorithm firstly identifies whether the current instruction is any of `block`, `loop`, or `if` (L6 to L8 in Algorithm 1) to recursively build AST. This is because only sub-trees led by these instructions can be nested. Then, the algorithm constructs the AST according to the stack type, as shown from L9 to L18 in Algorithm 1. For example, the first instruction, i.e., `i32.const 5`, takes no elements from the stack, and thus it jumps over the iteration at L10 and is pushed to *ASTs* at L17. Then, the following `local.set 0` takes an element from the stack as shown in Table 1. Thus, the while-loop at L10 is executed once. Within the while-loop, the last AST node, i.e., the previous `i32.const 5` is set as the child for `local.set 0`, which will then be pushed to *ASTs*. When the algorithm meets `call`, an instruction with a variable number of arguments, its context is determined in §3.2.1 by retrieving the function signature of the callee, and thus the AST can be built without any issue. Due to the recursive construction as shown at L7 in Algorithm 1, the `block` (L7 of Fig. 4b) will lead the `loop` (L8 of Fig. 4b), within which it implements the factorial calculation (L5 of Fig. 4a). Consequently, after traversing the `factorial` function, *ASTs* is composed of four nodes, each of which can be regarded as a root of a sub-tree of the AST (Fig. 4c).

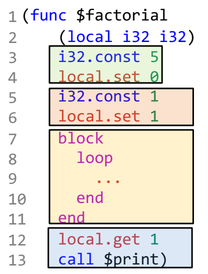
3.2.3 *Post-processing.*

The aim of the post-processing stage is to increase the diversity of the generated Wasm binaries. As we can see from Fig. 4c, the 1st and 2nd sub-trees are quite similar. In other words, if we take both of them into valid elements in the final corpus, the Wasm binaries with similar functions may be generated. Therefore, in the post-processing stage, we try to minimize the size of the corpus to avoid generating similar or even duplicated Wasm binaries. Specifically, we compare the newly extracted ASTs with the existing ones in the corpus, the implementation of which is shown in Listing 2. As we can see, the body is a two-layer nested loop at L19 that discards the AST that cannot increase the diversity through `compareAST`. The `compareAST` function recursively fetches children nodes and compares only the opcode of instructions. Take Fig. 4c as an example. When only considering the opcode, the 1st and 2nd ASTs are identical. Thus, only one of them will be kept.



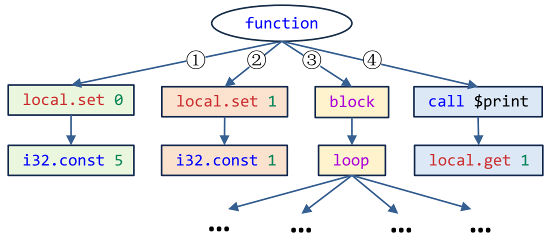
```
1 void factorial(){
2     int n = 5;
3     int result = 1;
4     for (int i = 1; i <= 5; i++){
5         result *= i;
6     }
7     print(result);
8 }
```

(a) The source code in C.



```
1  (func $factorial
2     (local i32 i32)
3     i32.const 5
4     local.set 0
5     i32.const 1
6     local.set 1
7     block
8       loop
9         ...
10      end
11    end
12    local.get 1
13    call $print)
```

(b) Compiled bytecode.



(c) The parsed AST of the Wasm bytecode.

Figure 4: A concrete example of AST parsing.

```
 1 function postprocessing(ASTs):
 2
 3   function compareAST(new, old):
 4     # compare the opcode of current instruction
 5     if new.instr.opcode != old.instr.opcode:
 6       return False
 7     # compare the number of children nodes
 8     if len(new.subNodes) != len(old.subNodes):
 9       return False
10     # recursively compare their children nodes
11     for i in range(len(new.subNodes)):
12       if !compareAST(new.subNodes[i], old.subNodes[i]):
13         return False
14     return True
15
16   # get existing ASTs from corpus
17   existingASTs = getExistingASTs()
18   # remove deduplicate AST
19   for AST in ASTs:
20     for existingAST in existingASTs:
21       if not compareAST(AST, existingAST):
22         ASTs.remove(AST)
23         break
24   return ASTs
```

Listing 2: The pseudocode of `PostProcessing`.

3.3 Binary Generation & Mutation

Given the sub-trees extracted from ASTs of real-world Wasm binaries as corpus, as well as the extracted context information on each instruction, WASMaker can generate valid Wasm binaries efficiently and effectively. Additionally, WASMaker also conducts

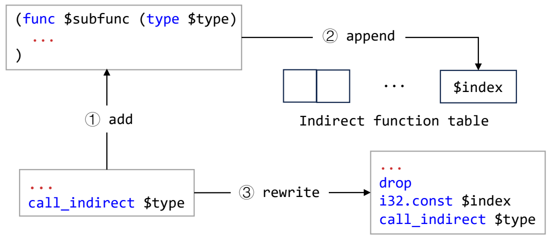


Figure 5: The concrete steps of WASMaker on maintaining necessary invoking relation of the callindirect instruction.

mutations on these generated binaries to bring in more diversities. The *binary generation* and *binary mutation* are detailed as follows.

3.3.1 *Binary Generation.* We generate a Wasm binary in a *bottom-up* way, i.e., *building a valid entry function*, *maintaining necessary invoking relation*, and *supplementing extra semantics*.

Step I: Building a valid entry function. First of all, we need to generate a function body for an entry function. In general, we randomly sample a specified number of AST sub-trees from the corpus. Then, we concatenate and transform them into a sequence of Wasm instructions, which is regarded as the body of the entry function. In Wasm, a function is composed of not only a set of instructions as its implementation, but also some local variables that can be accessed within the current function. Thus, WASMaker adds five local variables for the entry function, typed as `i32`, `i64`, `f32`, `f64`, and `v128` (see §2.1) and indexed from 0 to 4, respectively. For each variable instruction in the newly generated entry function, WASMaker gets their stack type mentioned in Table 1 and conducts necessary rewriting on the immediate number. For example, if a `local.get` is bound an operand with type `i64`, its immediate number, i.e., the index, will be rewritten as 1. Finally, according to the signature of the entry function, WASMaker appends necessary `local.get` before the final `return` instruction to avoid them being optimized as dead code.

Step II: Maintaining necessary invoking relation. Except for considering the intra-functional validity when building the entry function, the inter-functional validity should also be ensured, i.e., maintaining function invoking relations from the entry function. In Wasm, only two instructions can invoke function calls, i.e., `call` and `call_indirect`. Thus, WASMaker traverses each instruction. Upon encountering a `call`, WASMaker generates a callee according to its bound concretized stack type and semantic constraint. Instead of generating a callee with an empty function body whose aim is solely to maintain the stack balance, the callee adopts the same method as we mentioned in the **Step I**. In other words, the callee has a functional function body, and it may recursively generate its callees. For `call_indirect`, maintaining its invoking relation requires a little extra effort. Specifically, the callee of `call_indirect` is determined at runtime. As shown in Fig 5, WASMaker firstly generates a callee as its handling on `call`. Then, it inserts the function index into the function table to ensure the indexing process raises no exceptions. Last, to make sure the `call_indirect` can actually be guided to the newly generated callee, WASMaker inserts a `drop` and an `i32.const` *c* before it, where the *c* is the index.

Step III: Supplementing extra semantics. For a Wasm binary, only focusing on functions cannot fully guarantee its semantic correctness because the functionalities are decoupled into all 13 sections. Thus, extra semantics should be supplemented. Take the *data* section as an example, which is used to initiate the linear memory of Wasm binary. To prevent memory-related instructions from raising exceptions, WASMaker goes through the memory instructions of each function to determine the maximum addressing range based on its semantic constraint. Within the range, WASMaker will fill random data into the *data section*. Similarly, for the *table* and *element* sections that make up the indirect function table of Wasm, we construct them based on the context of table instructions in the binary. This ensures correct executions on table instructions. As for the global section, whose elements can be accessed by all functions, we adopt the same strategy as inserting local variables in functions. That is, WASMaker inserts only five global elements with different types into the global section. Additionally, it modifies the immediate numbers of all `global.get` and `global.set` instructions to match their bound stack type, replacing them with the corresponding index.

3.3.2 *Binary Mutation.* Though the methods in §3.3.1 can generate a substantial number of Wasm binaries, there are two shortcomings. First, increasing the code coverage of tested runtimes by assembling ASTs from existing Wasm binaries is still difficult. second, the ongoing evolution of Wasm makes it hard to verify the implementation of runtimes on new features. For instance, SIMD instructions are newly introduced to handle vectors, and the number of SIMD instructions is 236, far more than the one defined in Wasm 1.0 [42].

To address these shortcomings, we propose mutation strategies to test Wasm runtimes and uncover hidden issues comprehensively. Generally speaking, the mutation strategies can be divided into *AST-level mutation* and *module-level mutation*. The former one can be integrated into the **Step I** and **Step II** in §3.3.1 and the latter one can be performed when conducting the **Step III** in §3.3.1.

AST-level mutation is to mutate the instructions of ASTs in the corpus we collected, and its purpose is to extend the semantics of AST, so as to further improve the code coverage of runtime testing.

- *Mutate immediate numbers.* Improper processes in corner cases are more likely to trigger bugs or even vulnerabilities [20, 27, 44]. Therefore, part of the mutation strategies is put on the immediate number of instructions. For example, for constant instructions, like `i32.const`, WASMaker tries to replace the original operand with the value near the boundary, e.g., $2^{32} - 1$. Furthermore, for memory load and store instructions, WASMaker attempts to mutate the offset and alignment arguments.
- *Mutate operators.* Mutating operators can also increase the diversity of ASTs. To support SIMD instructions, WASMaker conducts mutations on numeric and memory instructions to their corresponding SIMD ones with similar semantics. For example, in Fig 6, the original implementation (left side) is `i32.const` followed by an `i32.load`. After mutating both of them into SIMD instructions with similar semantics, the mutated implementation (right side) is `v128.const` followed by a `v128.load`. However, we can observe that the type of the element consumed by `v128.load` mismatches the one pushed by `v128.const`. Fortunately, Wasm specification provides a set of instructions to convert the element type [40]. Thus, WASMaker adds an extra `i32x4.splat` to convert `v128` to `i32` to make the mutated implementation valid. We further design a strategy to increase the diversity of

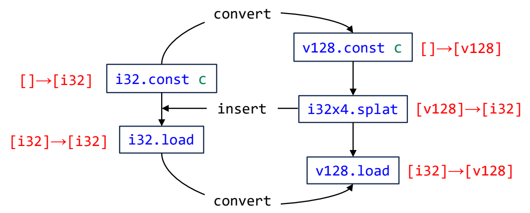


Figure 6: Mutating normal instructions to the SIMD ones.

ASTs. Specifically, as illustrated in Table 1, instructions may have identical stack type, like `i32.add` and `i32.sub`. Performing arbitrary interchanges among these instructions will not disrupt the stack balance. Consequently, WASMaker randomly replaces instructions with others based on the stack type.

Module-level mutation. In addition to the inconsistencies in the implementation of instructions, deficiencies in the validation of runtime on the whole Wasm module can also pose security risks. For example, incomplete memory boundary checks may lead to sandbox escapes. Thus, we also perform module-level mutations during the **Step III** in §3.3.1. Specifically, Cao et al. [8] have summarized five functionalities in Wasm, i.e., *Global*, *Import & Export*, *Memory*, *Function*, and *Custom*. Except for the last functionality, which has no association with the semantics of a Wasm binary, against each of the others, we have designed some specific mutation strategies, which are detailed in the following.

- *Global.* Except for the data type, each global variable possesses various other attributes, such as whether the variable is mutable. We randomly mutate the attributes of each global element.
- *Import & Export.* Wasm can import or export various semantics, such as functions, memory, and even global variables. WASMaker randomly adds import and export items (in the import and export sections) to the generated binaries.
- *Memory.* As we mentioned in the **Step III** in §3.3.1, the data section is initiated with random data. Except for that, Wasm still requires all memory accesses to be within the valid address limitation. Thus, WASMaker mutates the address limitation to test the correctness of memory boundary checks in runtimes.
- *Function.* Part of this functionality is related to the code section, which is handled by §3.3.1. Thus, WASMaker primarily mutates the table section. Similar to the memory section, the table section determines the range of the indirect function table, which is used for indexing by `call_indirect`. To this end, the correctness of boundary checks in runtimes can be covered.

3.4 Differential Testing

During differential testing, we first identify inconsistencies among runtimes. Given the large number of Wasm binaries tested, many inconsistencies may stem from the same underlying issue. Efficiently locating the root cause can enhance the overall testing process. Next, we explain how we address these issues.

3.4.1 Inconsistency Identification.

During the differential testing process, we *cannot simply take different outputs as inconsistency* because some of them are triggered by non-bug factors, like differences in design principles and implementational distinctions. First, Wasm is still under development, and thus runtimes may have different levels of supporting the Wasm specification. For example, some runtimes may not support SIMD instructions yet, resulting in an unsupported prompt when encountering Wasm binaries with SIMD instructions. Second, the output of various runtimes is also influenced by their implementation styles. For example, some runtimes may consider `i32` and `i64` integers as signed when outputting them, while other runtimes may treat them as unsigned. The Wasm specification does not explicitly define these behaviors. Third, different runtimes adopt diverse ways to handle error messages. For example, when encountering an out-of-bounds table access, some runtimes may only output *undefined element*, while other runtimes provide more specific reasons, such as *out of bounds table access*.

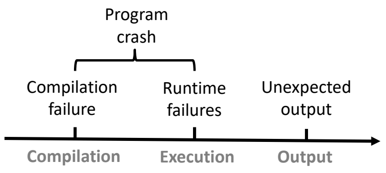


Figure 7: The types of inconsistencies during execution.

To reduce the number of false positives, i.e., inconsistent behaviors due to the above non-bug reasons, we give a clear definition of *inconsistent behavior* as shown in Fig. 7. Specifically, we define three kinds of inconsistent behaviors based on the lifetime of a Wasm binary being executed by runtimes, namely *compilation failure* (**CF**), *runtime failure* (**RF**), and *unexpected output* (**UO**). These three types are mutually exclusive and cover all inconsistent behaviors we identified during the differential testing. The first two types happen when runtimes encounter unexpected errors during the corresponding stage. As the name suggests, the compilation failure occurs when the Wasm binary is compiled and instantiated. The Wasm runtime will crash, that is, the input Wasm binary will not be executed at all. When Wasm binaries are executed, runtimes may still crash, usually due to raised exceptions, which is named runtime failure. We identify this inconsistent behavior by the classes of raised exceptions. The last one, i.e., unexpected output, can only be observed when a Wasm binary is executed.

To determine which runtime behaves unexpectedly, we take the majority rule as other differential testing studies adopt [46]. In other words, when executing a binary, we consider the behavior generated by most runtimes as correct. For example, if a binary encounters out-of-bounds memory access in three runtimes while the remaining runtime raises an indirect function table-related error, the latter one is considered potentially buggy.

3.4.2 Root Cause Localization.

Different Wasm binaries may trigger the same bug in Wasm runtimes, and thus performing an error localization is necessary to deduplicate these Wasm binaries according to the root causes of triggered inconsistency. To this end, we propose a *runtime-agnostic binary instrumentation method for root cause localization*, which is detailed in Algorithm 2, consisting of *function-level localization* and *instruction-level localization*.

Function-level Localization. It is designed for *runtime failure* and *unexpected output* inconsistency types (see §3.4.1). Specifically, the localization is implemented by static binary instrumentation. Before and after each `call` instruction, the instrumented function is responsible for printing the index of the invoked function along with its parameters or return values, which is achieved by the imported `fd_write` function, one of the WASI functions (see §2.1).

Formally, the `FuncLocalization` in Algorithm 2 illustrates the implementation of function-level localization. It takes the binary that exhibits inconsistencies and its corresponding inconsistency type as inputs. Then the binary is instrumented at the function level and then executed by runtimes to obtain the respective output logs (L2). At the loop at L4, the algorithm iterates over the logs outputted by instrumented functions, consisting of arguments and return values. The algorithm then compares the logs among runtimes to find the function that causes the inconsistent behavior (L5-L7). It is worth noting that the binary generation method guarantees each function is only called once and there is no recursive invocation. Thus, we can find the inconsistent function with a simple queue traversing (L6). Lastly, the algorithm examines whether the instruction-level localization should be further invoked depending on whether the inconsistency type is an unexpected output.

Algorithm 2 The error localization algorithm.

```
Input: binary - inconsistent binary, type - inconsistent type
Output funcid - the index of inconsistent function. instr - the
    inconsistent instruction
 1: function FuncLocalization(binary, type)
 2:     logsList ← FuncInstrumentation(binary)
 3:     funcid ← None
 4:     for each logs ∈ logsList do
 5:         if CompareFuncLogs(logs) = False then
 6:             funcid ← FindInconsistentFunc(logsList)
 7:         end if
 8:     end for
 9:     if type = OUTPUT then
10:         instr ← InstrLocalization(binary, funcid)
11:         return instr
12:     else
13:         return funcid
14:     end if
15: end function
Input: binary - inconsistent binary, funcid - the index of incon-
    sistent function
Output instr - The inconsistent instruction
 1: function InstrLocalization(binary, funcid)
 2:     logsList ← InstrInstrumentation(binary, funcid)
 3:     instr ← None
 4:     for each logs ∈ logsList do
 5:         if CompareInstrLogs(logs) = False then
 6:             instr ← logs.instr
 7:             break
 8:         end if
 9:     end for
10:     return instr
11: end function
```

Instruction-level Localization. The `InstrLocalization` will be invoked if the inconsistent behavior is due to the unexpected output. Specifically, it takes the binary exhibiting inconsistencies and the index of its inconsistent function as input. Then `InstrLocalization` performs instruction-level instrumentation on the specified function within the binary and feeds the instrumented binary to runtimes to get their output logs (L2). The instrumentation is conducted by printing the opcode of non-control-flow instruction and the value on the top of the stack after each instruction. L4 to L10 iterate the output of each instrumentation point. By comparing the values on the stack after the execution of each instruction, the algorithm identifies the specific instruction responsible for the observed inconsistent behavior. Consequently, the instruction that leads to buggy inconsistency is returned.

Re-run Strategy. It is important to note that multiple bugs can exhibit the same type of inconsistent behavior in runtimes. For example, if a runtime has bugs in both the implementation of memory-related instructions and the validation of memory boundaries, it may throw the out-of-bounds exception in both cases. To try to avoid such conflation issues, we employ the *re-run strategy*. That is, we re-run all binaries that exhibited inconsistent behaviors after fixing any of the identified bugs in a runtime. The goal is to discover additional unique runtime bugs through this process.

4 Implementation & Evaluation

4.1 Implementation & Experimental Setup

We have implemented WASMaker with over 7.4K LOC of Python3 code from scratch. All experiments were performed on a server in Ubuntu 22.04 with a 64-core AMD EPYC 7713 CPU and 256GB RAM. Our evaluation is driven by the following research questions:

RQ1 How effective is WASMaker compared to baselines?
RQ2 How many real-world bugs can be identified by WASMaker?
RQ3 What are the characteristics of the detected bugs?

Benchmark. We use the WasmBench [35], a well-known benchmark consisting of over 8K Wasm binaries. These binaries are collected from various sources, including code repositories, web applications, and package managers. Consequently, it guarantees the richness in terms of the semantics of Wasm binaries, laying the foundation for WASMaker to generate a corpus.

Baselines. We select wasm-smith [3] and WADIFF [46] as baselines. Specifically, wasm-smith is a Wasm binary generator that is widely adopted in testing Wasm runtimes [37, 38]. It is worth noting that Wasm binaries generated by wasm-smith in the default mode are mostly unable to be directly executed by runtimes because they may import functions that are not supported by runtimes. Therefore, we configure it to generate Wasm binaries without imported functions, and each binary has a minimum of ten functions, all of which are exported. As for WADIFF, it adopts symbolic execution on specifications of each instruction to generate Wasm binaries for testing the correctness of Wasm runtimes on executing instructions.

Targeted Runtime. We select representative runtimes according to two criteria. First, the stars of runtimes are greater than 3K. Second, the runtime is actively maintained for the last three months and has been officially released for over a year. Consequently, Wasmtime [38], Wasmer [37], WAMR [34], and WasmEdge [36] are selected. Note that no inconsistent behaviors are identified for Wasmer, and we omit it in the following three RQs.

Experimental Setup. To ensure a fair comparison, it is important to generate and execute as many Wasm binaries as possible within a 24-hour time budget using different tools and runtimes. During this process, we record any instances that lead to inconsistent behaviors. Note that the following root cause localization, runtime bug fixing, and re-running processes mentioned in §3.4 for identifying unique bugs in runtimes are not included within the 24-hour limit.

Table 2: The number of generated Wasm binaries and the ones that can trigger inconsistent behaviors, where CF, RF, UO, and T refer to compilation failure, runtime failure, unexpected output, and the sum of these cases, respectively. B and UB represent bugs and unique bugs that are not discovered by the other two tools.

		WASMaker	WASMaker_A	WASMaker_M	wasm-smith	WADIFF
# Generated Wasm Binaries		269,998	281,456	29,5269	343,299	283,330
Wasmtime	CF	3,151	0	2,548	0	0
	RF	0	0	0	0	0
	UO	0	0	0	0	0
	T	3,151	0	2,548	0	0
	B	1	0	1	0	0
	UB	1	0	1	0	0
WAMR	CF	40,158	36,314	38,662	5,555	0
	RF	7,578	4,138	6,274	5,479	0
	UO	1,441	983	0	0	443
	T	49,177	41,435	44,936	11,034	443
	B	7	4	2	5	2
	UB	4	4	0	4	0
WasmEdge	CF	0	0	0	0	0
	RF	10,032	7,647	3,918	0	0
	UO	3,543	1,414	0	0	0
	T	13,575	9,061	3,918	0	0
	B	7	5	3	0	0
	UB	7	5	3	0	0
Total (Sum of above runtimes)	CF	43,309	36,314	41,210	5,555	0
	RF	17,610	11,785	10,192	5,479	0
	UO	4,984	2,397	0	0	443
	T	65,903	50,496	51,402	11,034	443
	B	15	9	6	5	2
	UB	12	9	4	4	0

4.2 RQ1: Comparison with Baselines

Overall Result. Table 2 illustrates the overall results, where the 2nd, 5th, and 6th columns correspond to WASMaker, wasm-smith, and WADIFF, respectively. Moreover, the 2nd row illustrates the number of generated Wasm binaries, and the rows leading by **T** refer to the number of binaries that can result in inconsistent behaviors. As we can see, the efficiency of these three tools is indistinguishable, and they all can generate and perform the corresponding differential testing on roughly 300,000 Wasm binaries within 24 hours. However, when it comes to the number of Wasm binaries that can lead to inconsistent behaviors, the difference in efficiency among them is significant. WASMaker generates more than 65K Wasm binaries that can result in inconsistent behaviors, which is 6.0x and 148.8x greater than wasm-smith and WADIFF, respectively. Interestingly, we can easily observe that wasm-smith and WADIFF can only detect inconsistent behaviors in WAMR, while WASMaker is also effective in Wasmtime and WasmEdge. This is because the aim of binaries generated by wasm-smith is more about testing the syntax validation of runtimes, which may have been widely tested by the other runtimes before release. As for WADIFF, it can only test the implementation inconsistency at the instruction level.

Semantic Richness. Against these tools, we further evaluate the semantic richness of the generated binaries based on four metrics: nesting depth of code blocks (**ND**), average number of instructions (**ANI**), average number of control instructions (**ANCI**), and the ratio of actually executed instructions out of all instructions (**REI**).

Fig. 8 illustrates the distribution of **ND** for binaries generated by the three tools. It can be observed that when **ND** is smaller than 13, the curve of WASMaker is significantly higher than the other two tools, while wasm-smith surpasses WASMaker after that

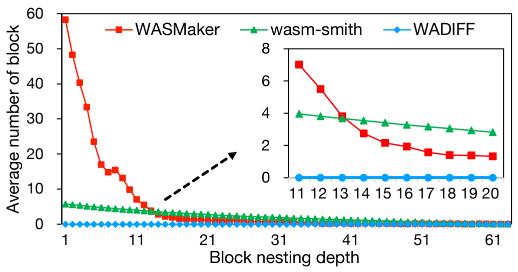


Figure 8: The ND distribution for generated Wasm binaries.

Table 3: ANI, ANCI, and REI of binaries generated by WASMaker, wasm-smith, and WADIFF.

Metrics	WASMaker	wasm-smith	WADIFF
ANI	15948.92	2568.73	5.26
ANCI	594.07	216.17	0.00
REI	39.66%	5.55%	100.00%

point. This indicates wasm-smith is good at generating complex code block patterns compared to WASMaker. We speculate this is because WASMaker generates binaries by combining ASTs of existing binaries, and thus, the depth of code blocks is more similar than the one of real-world binaries. On the other hand, wasm-smith is designed to validate the syntax-checking process of runtimes, thus intentionally generating such complex code block structures. Moreover, WADIFF does not generate test cases specifically targeting control instructions, indicating the **ND** is always 0. Table 3 presents the results of the other three metrics. As we can see, both **ANI** and **ANCI** of WASMaker are higher than the ones of the other two baselines. As for **REI**, because WADIFF does not generate nested code blocks, as shown in Fig. 8, all instructions in generated binaries will be executed once. Moreover, though WASMaker exhibits a weaker capability in generating deeply nested code blocks compared to wasm-smith, its **REI** is much higher. Therefore, we can conclude that code blocks generated by WASMaker are more efficient in testing runtimes. In summary, compared to the baselines, the binaries generated by WASMaker exhibit richer semantics.

Distribution of Inconsistencies. We further analyze the types of inconsistencies (mentioned in §3.4.1). Interestingly, for Wasmtime, all inconsistencies are **CF**, whereas for WasmEdge, it was the other way around. We speculate this is due to the different robustness of the two runtimes in their implementations. Moreover, most cases that can lead to inconsistency can be found in WAMR for all three tools, while only WASMaker can identify all three kinds of inconsistencies, further strengthening its effectiveness.

Bugs. We take a step further to investigate the root causes of these inconsistent behaviors, as shown in the rows led by **B** and **UB**, indicating the number of bugs and unique bugs that are not discovered by the other two tools. WASMaker identifies 15 bugs in total within 24 hours, while 12 of them are not discovered by wasm-smith and WADIFF. Interestingly, we can also observe that wasm-smith found four unique bugs that are not identified by WASMaker during testing against WAMR, which are all syntactic bugs. For example, one case has a `br_table` followed by an array with more than 30 branches as targets, where WAMR failed to examine whether the type of each targeted block is identical. Such a long `br_table` is unusual in real-world cases, and thus WASMaker cannot find this issue within 24 hours even with the help of mutation strategies.

Table 4: The breakdown of 167,343 Wasm binaries that can lead to inconsistent behaviors generated by WASMAKER. The number in parenthesizes is the number of unique bugs.

	CF	RF	UO	Total
Wasmtime	8,692 (1)	0 (0)	0 (0)	8,692 (1)
WAMR	99,578 (4)	17,568 (10)	2,763 (5)	119,909 (19)
WasmEdge	0 (0)	30,273 (7)	8,469 (6)	38,742 (13)
Total	108,270 (5)	47,841 (17)	11,232 (11)	167,343 (33)

Ablation Study. To evaluate the effectiveness of AST-level and module-level mutation strategies (see §3.3.2), we conducted an ablation study. Specifically, following the same experimental setup, we launch WASMAKER$_A$ and WASMAKER$_M$ separately, where only the AST-level and module-level mutation strategies are adopted, respectively. The 3rd and 4th columns of Table 2 illustrate the results. As we can see, WASMAKER$_M$ fails to uncover inconsistent behaviors under the **UO** category. This is because the module-level mutations put attention on sections other than the code section, where instructions are included. Additionally, the AST-level mutation demonstrates superior effectiveness as it directly targets the critical aspect of runtimes, i.e., the implementation of instructions. Note that, both WASMAKER$_A$ and WASMAKER$_M$ perform worse than WASMAKER, which enables two strategies simultaneously, indicating that the combination of AST- and module-level mutation strategies can yield better results.

Answer to RQ1. *Compared with the state-of-the-art baselines, i.e., wasm-smith and WADIFF, WASMAKER demonstrates its enhanced semantic richness, along with superior efficiency and effectiveness, in identifying unexpected behaviors in Wasm runtimes. It outperforms them by identifying 6.0x and 148.8x more cases, respectively.*

4.3 RQ2: Identified Real-World Runtime Bugs

Our previous exploration suggests that WASMAKER outperforms state-of-the-art techniques greatly. Thus, we next seek to apply WASMAKER to identify runtime bugs in the real world. For the targeted runtimes, WASMAKER performs differential testing for 72 hours[1]. In total, WASMAKER has generated 832,053 Wasm binaries, 167,343 out of which (20.11%) can lead to inconsistent behavior across Wasm runtimes. Table 4 illustrates the breakdown details.

Behavior Inconsistencies. Among all these three runtimes, WASMAKER generates a total of 119,908 cases that can trigger inconsistencies in WAMR, which is 3.1x and 13.8x greater than WasmEdge and Wasmtime. Most of the inconsistent cases belong to **CF**, accounting for 65% of all binaries.

Bugs. As shown in Table 4, for all the 167,343 inconsistent cases, our root cause localization method pinpoints 33 unique bugs, and 5, 17, and 11 of them are classified into **CF**, **RF**, and **UO**, respectively. Over half of the bugs are discovered within the first 24 hours, and the growth rate gradually slows down in the following hours. Note that, against Wasmtime, we only identified a single type of bug that can generate its unexpected behaviors, which is consistent with the results presented in RQ1. It is noteworthy that although WASMAKER generated the highest number of binaries resulting in **CF**, there are only 4 unique bugs. After investigating the reason, we found that it is because WAMR raises exceptions when parsing the complex representation of immediate values of `v128.const`. Because our mutation strategies (see §3.3.2) can import such instructions, a large number of binaries trigger this inconsistent behavior.

[1]Note that, the reason we set the time frame of 72 hours is that the number of unique bugs WASMAKER identified remain stable after roughly 65 hours.

Table 5: Detailed information of all confirmed bugs.

Runtime	Issue	Type	Root Cause	Status
Wasmtime	#7558	**CF**	type conversion error	Fixed
WAMR	#2450	**RF**	integer overflow	Fixed
	#2555	**RF**	llvm compiler crash	Fixed
	#2556	**CF**	non-alignment issue	Fixed
	#2557	**CF**	value kind check missing	Fixed
	#2561	**UO**	instruction implementation error	Fixed
	#2677	**CF**	type conversion error	Fixed
	#2690	**UO**	instruction implementation error	Fixed
	#2720	**RF**	memory check error	Fixed
	#2789	**CF**	string check error	Fixed
	#2861	**UO**	instruction implementation error	Confirmed
	#2862	**UO**	instruction implementation error	Confirmed
WasmEdge	#2812	**UO**	instruction implementation error	Confirmed
	#2814	**RF**	memory check error	Confirmed
	#2815	**UO**	instruction implementation error	Confirmed
	#2988	**UO**	instruction implementation error	Confirmed
	#2996	**UO**	instruction implementation error	Confirmed
	#2997	**UO**	instruction implementation error	Confirmed
	#2999	**UO**	instruction implementation error	Confirmed
	#3018	**RF**	memory check error	Confirmed
	#3019	**RF**	memory check error	Confirmed
	#3057	**RF**	memory check error	Confirmed
	#3063	**RF**	memory check error	Confirmed
	#3068	**RF**	memory check error	Confirmed
	#3076	**RF**	memory check error	Confirmed

Bug Reporting. For each unique bug identified, we randomly select binaries and report them to runtime developers, and we further provide root cause analysis to them for aiding bug fixing. Out of the 33 unique bugs, 25 have been confirmed by developers, and 11 have been fixed by the time of this writing, as shown in Table 5. For those confirmed ones, we will conduct detailed case studies on some representatives in the following §4.4

Answer to RQ2. *WASMAKER successfully generated more than 167K Wasm binaries that can lead to unexpected behaviors of Wasm runtimes, which were caused by 33 unique bugs. With our timely disclosure, 25 bugs have already been confirmed by runtime developers, and 11 of them have been fixed with our help.*

4.4 RQ3: Bug Characterization

We go a step further to analyze the root cause of each bug manually. As shown in Table 5, most of these bugs are caused by instruction implementation errors, while some other bugs are due to boundary check and type conversion errors. We next select three fixed bugs as representative ones for case studies.

```
1 (memory (;0;) 65536 65536)
2 (data (;0;) (i32.const -79158787) "Bp222N")
```

Listing 3: Wasmtime Type conversion error.

Case 1: Wasmtime Type conversion error. Listing 3 shows a part of the Wasm binary generated by WASMAKER, leading to the #7558 issue of Wasmtime. Specifically, L1 specifies the maximum space of the linear memory of this binary is 4 GB. However, L2 initializes the memory area starting from the offset of -79158787, which needs to be interpreted as an unsigned 32-bit integer. Unfortunately, Wasmtime incorrectly takes this number as a signed one, leading to a compilation failure since a negative address is invalid. In contrast, other runtimes correctly convert -79158787 to 4215808509, and perform the following expected behaviors correctly.

```
1 (module
2  (func (result v128)
3  v128.const i32x4 0x3d52aa71 0xea2f90b2 0xb20cdf3d 0
       x4d6054bc
4  i32.const -7235
5  i8x16.shl)
6  (export "main" (func 0)))
```

Listing 4: WAMR instruction implementation error.

Case 2: WAMR instruction implementation error. Listing 4 illustrates a part of the Wasm binary generated by WASMaker that triggers the #2690 issue of WAMR. To be specific, the `i32.const -7235` at L4 determines how many bits should be left-shifted, and it should be interpreted as the unsigned number 58302. Unfortunately, WAMR incorrectly treats -7235 as a signed number and directly performs modulo with 128. Thus, the following instruction left shifts the `v128.const` at L3 in different bit. To this end, though this code snippet can be executed normally, WAMR outputs a different output compared with other runtimes.

```
(export "\00jCeH" (func 0))
(export "" (func 1))
(export "fj" (func 2))
```

Listing 5: WAMR string check error.

Case 3: WAMR string check error. Listing 5 shows a case that leads to the #2789 issue of WAMR. The first two export items export `\00jCeH` and an empty string, respectively. Note that, both of them start with a `\00`. Instead of using Unicode string to handle import and export names, as required by the Wasm spec, WAMR adopts c-style string. Thus WAMR mistakenly resolves the first two exports to two empty strings, which causes WAMR to fail to compile the bytecode and raise the *duplicate export name* exception.

Answer to RQ3. *Instruction implementation error and memory check error are the top two factors leading to the unexpected behaviors of Wasm runtimes, underlining the importance of performing adequate sanity checks and following the specification.*

5 Discussion

Ethical Considerations. This work aims to identify real-world bugs in Wasm runtimes. We have identified 33 unique bugs, and reported them to runtime developers immediately. As of this writing, most of them have been confirmed, although some bugs are in the process of being fixed. We have repeatedly urged developers to fix them by providing necessary help.

Semantics. Although WASMaker tries to generate binaries with as rich semantics as possible, they might not be as semantically rich as real-world binaries. This is mainly due to the fact that some complex control flows within sub-trees of ASTs cannot be fully executed. In order to access these deep and nested nodes, reliance is placed on the context provided by the preceding ASTs. Although we assign values to local variables in functions, we cannot guarantee that we can restore every context.

Scalability. Wasm is experiencing rapid development, and many new proposals are emerging. WASMaker's current ability to test the semantics of runtimes might not encompass some new proposals. However, we argue that WASMaker can be easily extended to cover the new semantics by incorporating real-world Wasm binaries that exploit new features. Additionally, we will design more generation and mutation strategies for these new semantics to expand the capabilities of WASMaker.

6 Related Work

Wasm Binary Security. Many studies focus on the security side of Wasm binaries [5, 15–17, 22–24, 27, 31]. Lehmann et al. [22] analyzed the memory issues in Wasm binaries, and adopted binary instrumentation and AFL to detect memory bugs [23]. Wasmati [5] and Wasp [24] leverage code property graphs and concolic execution, respectively, to identify vulnerabilities in Wasm binaries.

Wasm Runtime Testing. There is some work on testing Wasm runtimes [6, 18, 19, 44, 46]. Specifically, WADIFF [46] utilizes symbolic execution on the specification of instructions to generate Wasm binaries to conduct differential testing. Moreover, Zhang et al. [44] developed a pattern-based runtime bug detection framework based on pre-defined domain knowledge. Additionally, Jiang et al. [19] leveraged existing performance testing benchmarks to discover performance issues in server-side Wasm runtimes. It is worth noting that wasm-mutate [6] can mutate binaries to test runtimes, but its strategies are inefficient. For example, adding function signatures and dead functions can not trigger the semantic implementation parts of runtimes. Additionally, its instruction mutation that preserves the original functionality fails to cover all Wasm instructions as comprehensively as possible. Thus, these approaches still struggle to generate Wasm binaries with rich semantics.

Differential Testing. McKeeman [25] first introduced the concept of differential testing. Since then, it has been extensively employed to test various applications, including virtual machines [4, 9, 10], compilers [28–30], deep learning frameworks [11, 13], and symbolic execution engines [21]. This paper draws on the idea of differential testing and applies it to Wasm runtimes.

7 Conclusion

This paper presents WASMaker, a novel differential testing framework for Wasm runtimes. WASMaker can generate syntactically correct and semantic-rich Wasm binaries by disassembling and assembling real-world Wasm binaries. For further pinpointing the root causes of inconsistencies and locating the bugs, we propose a root cause identification algorithm that can accurately locate bugs on a function level or even an instruction level. WASMaker shows great effectiveness via extensive evaluation, and we have uncovered 33 real-world bugs in popular Wasm runtimes.

Data-Availability Statement

The artifact of WASMaker is released at [2, 7].

Acknowledgement

This work was supported in part by National Key R&D Program of China (2021YFB2701000), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research, HUST CSE-HongXin Joint Institute for Cyber Security and research grants from Huawei. Mu Zhang is supported by NSF OAC-2115167, DARPA HR00112120009 Cooperative Agreement, and a Cisco gift.

References

[1] 2023. Occam's razor. https://en.wikipedia.org/wiki/Occam%27s_razor

[2] 2024. WASMaker. https://github.com/security-pride/WASMaker.

[3] Bytecode Alliance. 2023. Github wasm-tools repository. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith

[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364.

[5] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745.

[6] Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, and Benoit Baudry. 2024. Wasm-Mutate: Fast and effective binary diversification for WebAssembly. *Computers & Security* 139 (2024), 103731.

[7] Shangtong Cao. 2024. *WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-aware Binary Generation.* https://doi.org/10.5281/zenodo.12670309

[8] Shangtong Cao, Ningyu He, Yao Guo, and Haoyu Wang. 2023. BREWasm: A General Static Binary Rewriting Framework for WebAssembly. In *International Static Analysis Symposium*. Springer, 139–163.

[9] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[11] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential Testing of Cross Deep Learning Framework {APIs}: Revealing Inconsistencies and Vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7393–7410.

[12] eosio. 2023. eosio official website. https://eos.io/

[13] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 486–498.

[14] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[15] Keno Haßler and Dominik Maier. 2021. Wafl: Binary-only webassembly fuzzing with fast snapshots. In *Reversing and Offensive-oriented Trends Symposium*. 23–30.

[16] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts.. In *USENIX Security Symposium*. 1271–1288.

[17] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. 2023. Eunomia: Enabling User-specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries. *arXiv preprint arXiv:2304.07204* (2023).

[18] Bo Jiang, Zichao Li, Yuhe Huang, Zhenyu Zhang, and W Chan. 2022. Wasmfuzzer: A fuzzer for webassembly virtual machines. In *34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022*. KSI Research Inc., 537–542.

[19] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R Lyu. 2023. Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing. *arXiv preprint arXiv:2309.12167* (2023).

[20] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2940–2955.

[21] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 590–600.

[22] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of webassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 217–234.

[23] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. *arXiv preprint arXiv:2110.15433* (2021).

[24] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[25] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[26] MDN. 2023. MDN web docs website. https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm

[27] Alexandra E Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, et al. 2023. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 425–454.

[28] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model. *ACM SIGPLAN Notices* 48, 6 (2013), 187–196.

[29] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: differential testing for embedded compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. 21–30.

[30] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience* 37, 14 (2007), 1475–1488.

[31] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. 2022. Security risks of porting c programs to WebAssembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. 1713–1722.

[32] TinyGo. 2023. TinyGo official docs webpage. https://tinygo.org/docs/guides/webassembly/

[33] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4.

[34] wamr. 2023. Github wamr repository. https://github.com/bytecodealliance/wasm-micro-runtime

[35] WasmBench. 2023. Github WasmBench repository. https://github.com/sola-st/WasmBench

[36] WasmEdge. 2023. Github WasmEdge repository. https://github.com/WasmEdge/WasmEdge

[37] Wasmer. 2023. Github Wasmer repository. https://github.com/wasmerio/wasmer

[38] wasmtime. 2023. Github wasmtime repository. https://github.com/bytecodealliance/wasmtime

[39] WebAssembly. 2023. Github WASI repository. https://github.com/WebAssembly/WASI

[40] WebAssembly. 2023. Index of WebAssembly instructions. https://webassembly.github.io/spec/core/appendix/index-instructions.html

[41] WebAssembly. 2023. SIMD proposal for WebAssembly. https://github.com/WebAssembly/simd

[42] WebAssembly. 2023. WebAssembly 1.0 specification webpage. https://www.w3.org/TR/wasm-core-1/#a7-index-of-instructions

[43] WebAssembly. 2023. WebAssembly specification webpage. https://webassembly.github.io/spec/core/binary/index.html

[44] Zhang Yixuan, Cao Shangtong, Wang Haoyu, Zhenpeng Chen, Luo Xiapu, Mu Dongliang, Ma Yun, Huang Gang, and Liu Xuanzhe. 2023. Characterizing and Detecting WebAssembly Runtime Bugs. *ACM Transactions on Software Engineering and Methodology* (2023).

[45] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 301–312.

[46] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2023. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 939–950.

Received 2024-04-12; accepted 2024-07-03