Characterizing and Detecting WebAssembly Runtime Bugs

YIXUAN ZHANG, Peking University, China
SHANGTONG CAO, Beijing University of Posts and Telecommunications, China
HAOYU WANG, Huazhong University of Science and Technology, China
ZHENPENG CHEN, University College London, UK
XIAPU LUO, The Hong Kong Polytechnic University, China
DONGLIANG MU, Huazhong University of Science and Technology, China
YUN MA, Peking University, China
GANG HUANG, Peking University, China
XUANZHE LIU, Peking University, China

WebAssembly (abbreviated WASM) has emerged as a promising language of the Web and also been used for a wide spectrum of software applications such as mobile applications and desktop applications. These applications, named as WASM applications, commonly run in WASM runtimes. Bugs in WASM runtimes are frequently reported by developers and cause the crash of WASM applications. However, these bugs have not been well studied. To fill in the knowledge gap, we present a systematic study to characterize and detect bugs in WASM runtimes. We first harvest a dataset of 311 real-world bugs from hundreds of related posts on GitHub. Based on the collected high-quality bug reports, we distill 31 bug categories of WASM runtimes and summarize their common fix strategies. Furthermore, we develop a pattern-based bug detection framework to automatically detect bugs in WASM runtimes. We apply the detection framework to seven popular WASM runtimes and successfully uncover 60 bugs that have never been reported previously, among which 13 have been confirmed and 9 have been fixed by runtime developers.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: WebAssembly, WebAssembly runtime

ACM Reference Format:
Yixuan Zhang, Shangtong Cao, Haoyu Wang, Zhenpeng Chen, Xiapu Luo, Dongliang Mu, Yun Ma, Gang Huang, and Xuanzhe Liu. 2023. Characterizing and Detecting WebAssembly Runtime Bugs. 1, 1 (September 2023), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

WebAssembly (abbreviated WASM) has quickly emerged as a promising language of the Web in recent years [56]. WASM is a binary instruction specification [56, 61, 72] for a stack-based virtual machine and provides developers with an equivalent textual format [38] for reading, testing, learning

Authors' addresses: Yixuan Zhang, Peking University, Beijing, China, zhangyixuan.6290@pku.edu.cn; Shangtong Cao, Beijing University of Posts and Telecommunications, Beijing, China, shangtongcao@bupt.edu.cn; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn; Zhenpeng Chen, University College London, London, UK, zp.chen@ucl.ac.uk; Xiapu Luo, The Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk; Dongliang Mu, Huazhong University of Science and Technology, Wuhan, China, dzm91@hust.edu.cn; Yun Ma, Peking University, Beijing, China, mayun@pku.edu.cn; Gang Huang, Peking University, Beijing, China, hg@pku.edu.cn; Xuanzhe Liu, Peking University, Beijing, China, liuxuanzhe@pku.edu.cn.

instructions, and debugging, etc. Although WASM was initially proposed for Web applications [70, 73], it is moving fast towards a much wider spectrum of domains, including desktop applications [24, 39], mobile applications [24], IoT [64, 75], blockchain [18, 19, 46], serverless computing [24, 55], and edge computing [54, 66], etc. To develop these applications (named WASM applications), developers can compile high-level programming languages to WASM binaries or convert the equivalent manually-written textual format to WASM binaries. WASM binaries are commonly executed in WASM runtimes. A WASM runtime provides an efficient, memory-safe, sandboxed execution environment for WASM applications [42]. However, a great variety of WASM runtime specific bugs have been reported by developers, inevitably impeding the development of the WASM application ecosystem. Despite this, WASM runtime bugs have not been systematically studied by our community. Therefore, there is a general lack of an understanding of these bugs, including their root causes, fix patterns, and how to detect these bugs in emerging WASM runtimes.

This Work. To fill in the knowledge gap, we present the first comprehensive study on characterizing and detecting bugs in WASM runtimes. We focus our study on three most popular and representative WASM runtimes, including `wasmtime` [36], `wasmer` [33], and `WebAssembly Micro Runtime(WAMR)` [40]. We first collect 903 bug-related posts from GitHub, a commonly-used data sources for studying software bugs, and make an effort to identify 311 real-world bugs of these WASM runtimes (see Section **3**). Based on the collected bugs, we manually construct a taxonomy of 31 bug categories (see Section **4**), indicating the diversity of WASM runtime bugs. Moreover, we summarize common fix patterns for each bug category (see Section **5**). These empirical results provide a high-level categorization that can serve as a guide for developers to resolve common faults and for researchers to develop tools for detecting and fixing common WASM runtime bugs.

Furthermore, we develop a pattern-based bug detection framework based on the knowledge summarized from the bug taxonomy, to test the presence of bugs in WASM runtimes (see Section **6**). To evaluate the generalizability of our study, beyond the three analyzed WASM runtimes, we further consider four emerging WASM runtimes (`wasm3`, `WASMEdge`, `wasmer-go`, and `wasmer-python`) for bug detection. We have successfully identified 60 previously-unknown bugs. We report these bugs to the developers of corresponding WASM runtimes. By the time of this writing, 13 bugs have been confirmed by the developers, and 9 of them have been fixed based on our suggestions.

To summarize, this paper makes the following contributions:

- We conduct the first systematic study of bugs in WASM runtimes. We summarize common bug categories and their corresponding fix strategies. Our results can help understand and characterize bugs in WASM runtimes while shedding light on future WASM-related studies.
- We develop a pattern-based bug detection framework based on the knowledge summarized from bug categories we created to automatically detect bugs in WASM runtimes. By applying the detection framework to real-world WASM runtimes, it shows that our proposed framework can effectively detect bugs and provide useful information to facilitate bug diagnosis and fixing.
- We make the scripts, datasets, and the bug detector available [44] to the research community for other researchers to replicate and build upon.

2 BACKGROUND

In this section, we introduce WASM binaries, Wat format, execution of WASM binaries, and WASM runtime architecture.

2.1 WASM binaries

WASM is a low-level assembly-like language that is designed for efficient execution and compact representation. The WASM binary file is compact like Java class files and is saved with the `.wasm` suffix [78]. The WASM specification defines a conceptual stack virtual machine for most WASM instructions to work on, performing numbers' pop and push and leaving the result on the stack.

2.2 Wat format

Wat format is a pretty-printed textual format (i.e., `.wat`) [38] provided for developers, which can be used to learn the syntax, understand the WASM module, test WASM program, optimize applications, debug code, write WASM programs by hand, etc.

Developers and users can use the wabt [20] tool to translate WASM binaries to WASM textual format or vice versa. A module is the fundamental unit of code in WebAssembly, and it is represented as a tree of nodes that describe the module's structure and code. This structure is depicted using S-expressions, a simple and old textual format for representing trees. WebAssembly's tree is flat, mainly comprising lists of instructions. In binary and textual formats, the module serves as the building block of WebAssembly programs. Each node in the tree goes inside a pair of parentheses *(...)*. The simplest WASM module is as follows: *(module)*. All code in a Webassembly module is grouped into functions, which have the following structure: *(func <signature> <locals> <body>)*. The signature declares the function parameters and return values. The locals are declared with explicit types. The body is just a linear list of WASM instructions [38]. As shown in Figure 1, the function *add* in the WASM module accepts two i32 values as the parameters and returns the pulsed value. And the function *add* is exported with the name *func1*.

wat code :

```
(module
  (func $add (param $x i32) (param $y i32) (result i32)
    local.get $x
    local.get $y
    i32.add)
  (export "func1" (func $add)))
```

Fig. 1. An example of a wat format file.

2.3 Execution of WASM binaries

Before illustrating the execution process of WASM binaries, we first introduce some terms. **High-level language** means the programming languages developers used to develop applications [69], such as Java, Python, Go, Rust, C, C++, JavaScript, etc. **Native code** means the machine code that is compiled to run directly on a specific processor or computer architecture without needing an interpreter or virtual machine [78]. **WASM code** is the same as WASM binaries and represents the equivalent Wat format. **Frontend compiler** means the WASM compilers which could compile high-level languages into WASM binaries, such as Emscripten [43]. After illustrating related terms, we next show the execution of WASM binaries. As a binary instruction format, WASM is designed as a portable compilation target for high-level programming languages [42]. As shown in Figure 2, developers can use WASM compilers(frontend compiler) to translate high-level language programs to WASM binaries. There are dozens of compilers available to compile different source language programs to WASM binaries, such as AssemblyScript, Emscripten, Rustc/WASM-Bindgen, etc [69]. WASM can be executed at native speed [56] on a wide range of platforms. The tool for this critical process is a WASM runtime, an intermediate layer between the WASM binaries and the hardware platforms. A WASM runtime should consider the structure, operating system, and other differences between various platforms and provide a relatively secure execution environment for the WASM binaries. As shown in Figure 2, developers can create applications in high-level languages, compile them into WASM binaries [16, 69], and execute WASM binaries in WASM runtimes. Alternatively, they could develop simple WASM programs in the textual format, convert them to WASM binaries through `wabt` [20], and execute the binaries in WASM runtimes.

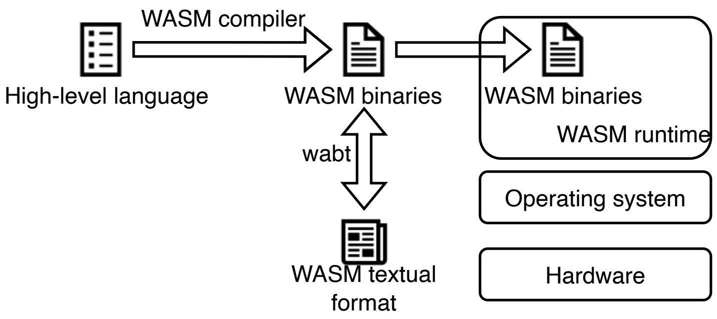


Fig. 2. The execution process of WASM binaries.

2.4 WASM Runtime Architecture

Based on the implementation of well known WASM runtimes [23, 25, 26, 32, 33, 36, 40], we have summarized the general architecture of WASM runtimes in Figure 3, which can be divided into six major components.

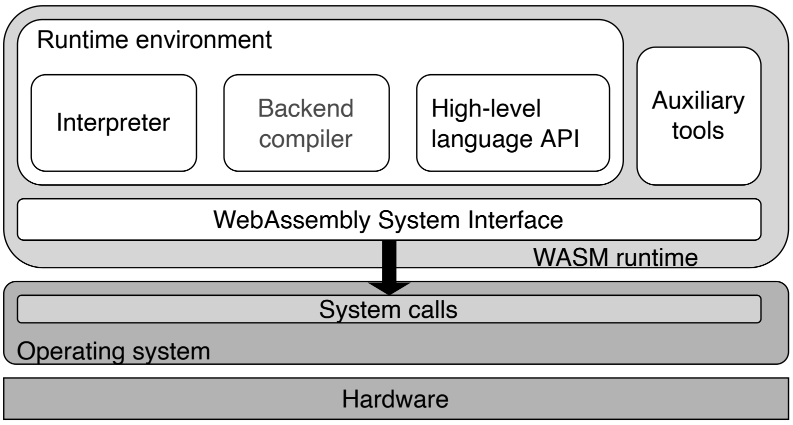


Fig. 3. The general architecture of a WASM runtime.

Backend compiler. WASM runtimes support executing WASM binaries in the following modes: interpreter mode, Ahead-of-Time compilation mode (AoT), and Just-in-Time compilation mode (JIT). WASM runtimes support compiling WASM binaries into native code before executing it locally using AoT compilers. To speed up the execution efficiency, some WASM runtimes use the just-in-time compilation of hot code through JIT compilers. JIT compilers and AoT compilers are considered backend compilers in the WebAssembly workflow.

Interpreter. Some WASM runtimes provide interpretive execution on the WASM binaries.

Runtime environment. The runtime environment supports allocating memory, performing stack operations, reporting execution error messages, and other features.

High-level language API. The WASM runtimes can be embedded in different high-level languages (e.g., C/C++, Java, Python, Rust, etc.) as a library to allow users to use WASM in any scenarios with various languages.

WebAssembly system interface. WASM runtimes provide WASM applications with WebAssembly system interface (WASI) [41] as a modular system interface [23], focusing on security and portability. WASI is the bridge between the sandbox environment and operating systems. WASI is an API that provides access to several OS-like features, including file operation and clock.

Auxiliary tools. WASM runtimes also provide handy little tools for the users, such as WASM module cache, WASM textual file format validation, etc.

3 STUDY DESIGN

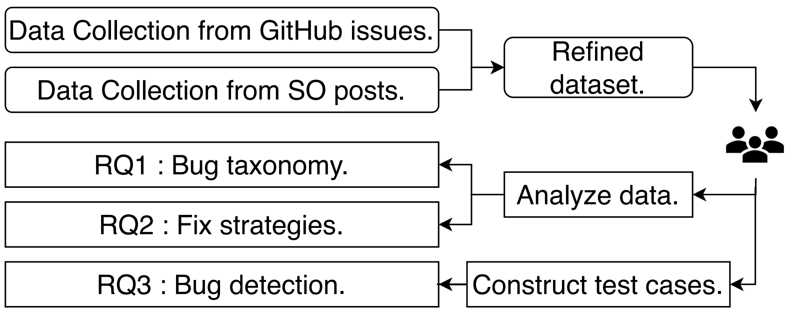


Fig. 4. Overview of the methodology.

3.1 Research Questions

We focus our analysis on the bugs in WASM runtimes. Specifically, we aim to address three research questions that are concerned with bugs in WASM runtimes:

RQ1 **Bug taxonomy.** What are the root causes of the bugs in WASM runtimes?
RQ2 **Fix strategies.** What are the common fix strategies for different bug symptoms?
RQ3 **Bug detection.** What is the effectiveness of the proposed bug taxonomy in uncovering bugs?

We first perform an empirical study to characterize WASM runtime bugs. Specifically, we seek to investigate: 1) **the taxonomy of bugs**, i.e., the reasons leading to the bugs, and 2) **the fix strategies**, i.e., how to address these bugs. Moreover, we construct the 3) **bug detection framework** to identify bugs in WASM runtimes. Figure 4 shows the overview of our study methodology.

3.2 Collection of WASM Runtime Bugs

To approach the answer, we collect and analyze the bug reports posted on **Github** and **Stack Overflow**, following the traditional empirical methods in the SE community [49, 53, 58, 68, 69, 71, 74, 77, 79, 80], as shown in Figure 4.

3.2.1 Selecting WASM Runtimes. As shown in Table 1, we select three most popular WASM runtimes as target, including `wasmer` [33], `wasmtime` [36], and `wasm-micro-runtime (WAMR)` [40]. We believe they are the most representative WASM runtimes for us to characterize real-world

Table 1. Statistics of our harvested dataset.

Runtime	Stars	Commits	Language	GitHub issues	SO posts	Total
wasmer	12,026	11,332	Rust	403 (179)	41 (0)	444 (179)
wasmtime	7,360	9,754	Rust	167 (94)	52 (0)	219 (94)
WAMR	2,720	686	C/C++	333 (38)	4(0)	337 (38)
Total				**903 (311)**	**97 (0)**	**1000 (311)**

*The refined numbers are in the parentheses.

WASM runtime bugs across different implementations, as 1) all of them are mature projects (with over 100,000 LOC) that have gained thousands of stars on GitHub, 2) they have covered different kinds of execution modes (i.e., Interpreter, JIT, and AoT), and 3) they are implemented in different languages (i.e., Rust and C/C++).

3.2.2 Data Collection from GitHub. Following previous work [49, 68, 69, 74, 79, 80], we extract issues in the official GitHub repositories of the selected WASM runtimes. GitHub issues contain a wealth of bug-related information, such as source code, comprehensive reports, and contributor discussions [51]. These characteristics make GitHub issues suitable for analyzing bug root causes and summarizing fix strategies. For details, we use the GitHub API shown in the artifact [44] to extract the related issues on May 14, 2022. GitHub issues include various topics, including bug reports, feature requests, documentation updates, etc. Thus, to highlight the purposes of bugs, we take advantage of the bug issue label to identify related issues. We collect issues related to `wasmer` and `wasmtime` by filtering labels with "bug". Due to all the issues from `WAMR` are not labeled, we extract all the issues from `WAMR` for further analysis. Overall, we obtain 403 issues from `wasmer`, 167 issues from `wasmtime`, and 333 from `WAMR`.

3.2.3 Data Collection from SO. Initially, we also consider posts from Stack Overflow. Each SO question has at least one tag based on its topics. We extract the posts related to the selected WASM runtimes on May 14, 2022. As a result, we obtain 41 posts for `wasmer`, 52 posts for `wasmtime`, and 4 posts for `WAMR`. Table 1 shows the collected raw data.

3.2.4 Refining the Dataset. We manually investigate the collected data from **GitHub** and **Stack Overflow**. First, we filter out GitHub issues and SO posts with no definite answers to ensure the accuracy and certainty of bugs and fix strategies. Second, we exclude installation/build bugs, documentation bugs, user misuse, and other issues and posts unrelated to WASM binaries' execution from the source data. Finally, as shown in Table 1, the total number of WASM runtime bugs is 311. The scale of this dataset is comparable and more extensive than those used in existing bug-related studies [45, 47, 49, 51, 69, 77, 79] that also require manual inspection. All the 311 issues are from GitHub since the 97 SO posts are all excluded in the data refining process. It is probably because there are few WASM experts on SO since WASM is an emerging language. Therefore, WASM developers tend to report the bugs they encounter to the official WASM runtime repositories to seek immediate help.

3.3 Labelling Bugs and Fix Strategies

The refined 311 bug reports are used for distilling features and fix strategies through manual labelling by two authors and an intercessor.

3.3.1 Pilot Labelling. First, we randomly sample 50% of the posts ($N = 155$) from the selected WASM runtimes for pilot labeling. The first two authors of the paper jointly participate in the process. According to the WASM runtime architecture and the root causes, they create the bug categories and fix strategies by analyzing the GitHub issues.

3.3.2 Reliability Analysis. For reliability analysis, the first two authors independently label the remaining 40% issues based on the taxonomy constructed in the prior stage. In detail, they mark each issue with the posted bug, fix strategy categories, and the issues that cannot be classified into the current taxonomies as a new category. To measure the reliability during the independent labelling, we employ the widely used Cohen's Kappa indicator (κ) for bug and fix strategies of 0.921 and 0.915, indicating almost perfect agreement [50]. The agreement levels demonstrate the reliability of our labelling.

The divergence in the labelling process is then discussed and settled after the labeling process. For the newly added categories by the first two authors, we discuss them with the intercessor. As a result, we add two new categories to the bug taxonomy and three new categories into the fix strategy taxonomy. Furthermore, the first two authors independently label the remaining 10% issues. During this process, no more bug taxonomy or fix strategy is added, indicating **saturation of the taxonomy**. After finishing the whole labelling stage, the Cohen's Kappa indicator (κ) for bug and fix strategies is 0.929 and 0.925, showing almost perfect agreement [50]. Additionally, the three authors involved in the taxonomy check the final labeling result together. We will detail the bugs and fix patterns in the following sections.

4 RQ1:TAXONOMY OF WASM RUNTIME BUGS

We present the hierarchical taxonomy of WASM runtime bugs according to the WASM runtime architecture (see Section 2). As shown in Figure 5, the taxonomy is organized into three-level categories, including a root category (*WASM Runtime Bugs*), four inner categories linked to different components in a WASM runtime (e.g., *Backend Compilation*), and 31 specific leaf categories (e.g., *Register allocation error*).

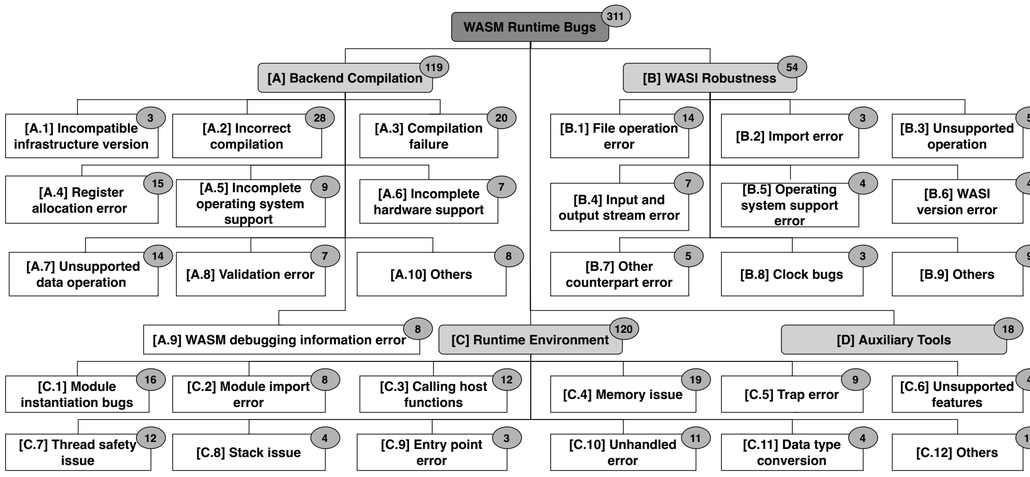

WASM Runtime Bugs 311
[A] Backend Compilation 119
[B] WASI Robustness 54
[A.1] Incompatible infrastructure version 3
[A.2] Incorrect compilation 28
[A.3] Compilation failure 20
[B.1] File operation error 14
[B.2] Import error 3
[B.3] Unsupported operation 5
[A.4] Register allocation error 15
[A.5] Incomplete operating system support 9
[A.6] Incomplete hardware support 7
[B.4] Input and output stream error 7
[B.5] Operating system support error 4
[B.6] WASI version error 4
[A.7] Unsupported data operation 14
[A.8] Validation error 7
[A.10] Others 8
[B.7] Other counterpart error 5
[B.8] Clock bugs 3
[B.9] Others 9
[A.9] WASM debugging information error 8
[C] Runtime Environment 120
[D] Auxiliary Tools 18
[C.1] Module instantiation bugs 16
[C.2] Module import error 8
[C.3] Calling host functions 12
[C.4] Memory issue 19
[C.5] Trap error 9
[C.6] Unsupported features 4
[C.7] Thread safety issue 12
[C.8] Stack issue 4
[C.9] Entry point error 3
[C.10] Unhandled error 11
[C.11] Data type conversion 4
[C.12] Others 18

Fig. 5. Taxonomy of bug symptoms. The number in the top right corner indicates the number of bugs for each category.

The backend compilers (JIT compilers and AoT compilers) of the architecture are summarized into one inner bug category, called *Backend Compilation (A)*, which converts WASM binaries into native code. The bugs in the lowest part in a WASM runtime are called *WASI Robustness (B)*. The handy little tools in WASM runtimes are called *Auxiliary Tools (D)*. Other bugs that occur while running WASM binaries are classified into *Runtime Environment (C)*, including memory allocation, calling host functions, and so on. It is worth mentioning that bugs that occur while using high-level language API are either divided into *Backend Compilation (A)* or *Runtime Environment (C)*. WASM users could use the API to compile WASM binaries and take advantage of functionalities in the Runtime Environment, and it is an interface for users to make good use of a WASM runtime. Moreover, the interpreter part is merged in a leaf category of *Backend Compilation (A)*, as only `WAMR` provides an interpreter, and only one bug is found in the interpreter.

4.1 Backend Compilation

As the first stage of executing WASM binaries, backend compilation is used to translate WASM binaries into native code. In general, backend compilers convert WASM binaries into their intermediate representation (IR), allocate registers and optimize the code. Note that backend compilers could convert WASM binaries into the IR proposed in other compilation framework infrastructures (e.g., LLVM). The whole process needs to support various OSs and CPU architectures. We observe 119 bugs in this category, accounting for 38.3% of all the classified bugs and covering 10 leaf categories.

Various backend compilers use their own IR as the intermediate step to translate WASM instructions to native code. During the process of compiling, compilers could generate incorrect IR or incorrect native code during the translation of WASM binaries. Besides, optimizing the code could also lead to an error. These bugs are summarized as *Incorrect compilation (A.2)*. A compiler may raise an exception when generating native code or even *fail to generate the native node (A.3)*, which accounts for 16.8% bugs in *Backend Compilation (A)*. Moreover, some WASM runtimes rely on the existing compilation framework, such as LLVM. Thus, *Using the incorrect version of infrastructure (A.1)* could lead to unexpected results, accounting for 2.5% bugs in *Backend Compilation (A)*.

Besides converting WASM instructions into native machine instructions, the backend compilers must allocate registers. However, they may result in the *Incorrect register allocation (A.4)*, including incorrectly using special registers, loading data from an unexpected register, and exhausting registers. These bugs account for 7.6% of bugs in *Backend Compilation (A)*. As shown in Example (a) (Figure 6), the allocation of `r15` poses a bug (lines 4,6,7 with highlight) [37]. Before describing this example, let us introduce two concepts. A control and status register (CSR) is a special type used to store control and status information about the processor or system. A pinned register refers to a register that is fixed or restricted in its use. It has a specific role and cannot be used for general-purpose operations like other registers. In `wasmtime`, the pinned register is enabled by the `enable_pinned_reg` Cranelift (the backend compiler) setting and used via the `get_pinned_reg` and `set_pinned_reg` CLIF ops. However, now the implementation of it is only supported when Cranelift is embedded in SpiderMonkey. In other cases, such as aarch64, Cranelift cannot correctly translate the usage of the pinned register in CLIF ops to native code, and Cranelift mistakenly uses *mov* ops to read and write the pinned register (r15). As another example in Example (b) (Figure 7), `wasmtime` allocates registers for the given wat file. However, the allocation of registers shows a bug while lowering Single Instruction Multiple Data (SIMD) instructions (line 5 with highlight in wat code) [15]. Vcode is a kind of intermediate representation (IR) used in `wasmtime` when lowering WASM binaries into native code. The `movdqa` instruction in Vcode is used to move the data from `v6` to `v7`, but `v6` is never set (line 6 with highlight in Vcode). This bug will cause panic during the execution of WASM binaries, and the execution process cannot be completed. Most WASM runtimes only support JIT or AoT compilation, while `WAMR` also provides an interpreter to deal with WASM. There is only one bug in the interpreter. The interpreter could not correctly pass parameters to submodules, leading to an incorrect result. Moreover, this is summarized into *Others (A.10)*.

Fault description : As shown in the Assembly code converting from wasmtime IR, r15, the pinned register, gets saved and restored as a CSR, making it impossible to use a pinned register.

Fault symptom : [A.4] Register allocation error

Assembly code :

```
   0:   55                      push    rbp
   1:   48 89 e5                mov     rbp, rsp
   4:   48 83 ec 10             sub     rsp, 0x10
   8:   4c 89 3c 24             mov     qword ptr [rsp], r15
   c:   4c 8b 0f                mov     r9, qword ptr [rdi]
   f:   49 83 c7 01             add     r15, 1
  13:   4c 8b 3c 24             mov     r15, qword ptr [rsp]
                                        //gets saved and restores as a CSR
  17:   48 83 c4 10             add     rsp, 0x10
  1b:   48 89 ec                mov     rsp, rbp
  1e:   5d                      pop     rbp
  1f:   c3                      ret
```

Fig. 6. Example (a) - GitHub `wasmtime` issue #4170

Fault description : Wasmtime convert the wat file into Vcode, as shown below. The *movdqa* instruction moves out of v6, which is never set.

Fault symptom : [A.4] Register allocation error

Fix strategy : Fix compilation rules

wat code :

```
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    v128.const i32x4 0x00000000 0x00000000 0x00000000 0x00000000
    i64x2.extend_low_i32x4_u
    v128.const i32x4 0x00000000 0x00000000 0x00000000 0x00000000
    i64x2.mul
    ......
  )
  ......
)
```

Vcode :

```
Block 0:
  (original IR block: block0)
  (instruction range: 0 .. 13)
  Inst 0:   movq    %rdi, %v0J
  Inst 1:   movq    %rsi, %v1J
  Inst 2:   movdqa  %v6V, %v7V    //moves out of v6, but v6 is never set
  Inst 3:   pxor    %v14V, %v14V
  ......
```

Fig. 7. Example (b) - GitHub `wasmtime` issue #3337

In the compilation process, WASM runtimes run the WASM file across various *operating systems (A.5)*. They account for 7.6% of bugs in the current inner category. The backend compilers encounter problems only caused by specific operating systems and lack consideration for their particular circumstances. With its architecture and instruction set, WASM runtimes also run the WASM files across different CPUs. Some problems are only present in specific CPUs or specific architecture machines. These problems are summarized as *Incomplete hardware support (A.6)* which account for 5.9% bugs in Backend Compilation.

Further, we observe 11.8% of bugs in *Unsupported data operation (A.7)*. As an example, the backend compiler Cranelift used in `wasmtime` does not support `srem.8` and `srem.16` operations in its intermediate representation (IR) [16, 17]. Although the WASM binaries can be translated into Cranelift IR using `srem.8` and `srem.16`, the implementation of these operations in Cranelift is currently incomplete, as reported in `wasmtime` issue #2826 [12]. Besides, the lowering from WASM binaries into cranelift IR in `wasmtime` could lack the design of supporting the data operation in big endianness machines (e.g., GitHub `wasmtime` issue #3288). Executing the clif file on s390 hardware shows wrong results only for i16, i32, and i64 types, while i8 passes these tests. The s390 architecture is big-endian, while the data operation in `wasmtime` was taken from the lower bites. Thus, the data operation of i16, i32, and i64 was not supported in the big-endianness machine. This kind of bug could pose different execution results or execution exceptions. This bug can result in inconsistent results or execution exceptions for the same WASM binaries executed on different machines. Besides, WASM specification introduced SIMD instructions to improve execution efficiency. Backend compilers may lack the support for data operation related to SIMD instructions, such as the operation of the v128 data type.

During the compilation process, the verifier must validate the legalization of IR, WASM instructions, and the temporary files emitted by AoT compilers. The *incorrectness or strictness of validation (A.8)* causes errors in the backend compilation process, accounting for 5.9% bugs in *Backend Compilation (A)*. To make it easier for the users to utilize the WASM runtimes and learn about the code error, the WASM runtime developers should consider the debugging information during the backend compilation. The *WASM debugging information (A.9)* bugs lead to several consequences, such as failing to provide debugging information or even influencing the compilation of WASM information, accounting for 6.7% bugs in *Backend Compilation (A)*.

4.2 WASI Robustness

WASI is the fundamental part of a WASM runtime, allowing WASM to run outside the web. WASI supports WASM with several OS-like features, including files, sockets and the clock. Each WASM runtime could implement its specific features. Bugs in this part account for 17.4% of our dataset.

WASI allows the WASM binaries to perform file operations, including making a new directory, writing and reading files, deleting files, and so on. The most common bug related to WASI is *File operation error (B.1)*, which accounts for 27.8%. For example, users failed to rename a file through WASI by applying `wasmer` in the `wasmer` issue #2297 [11]. Besides, *Input and output stream error (B.4)* and *Clock Bugs (B.8)* are also found in this part, accounting for 13.0% and 5.6% of bugs in WASI Robustness individually. Different WASM runtimes implement their own WASI, which may lack the *support for some operations (B.3)*. 9.3% of bugs in WASI Robustness are triggered due to unsupported operations. For example, in the `wasmer` issue 1640 [7], the author reported that when using the function *siglongjmp* in a WASM file, `wasmer` encounters an error. It is due to *setjmp* and *setlongjmp* are not supported in WASI. For another example, in the `wasmer` issue 1263 [4], the author reported that WASI syscalls lack support for pre-opened directories.

In addition to the basic functionalities, WASI relies on different WASI modules and versions. Frontend compilers convert the high-level language into WASM binaries which may include a few WASI modules and different versions. WASM should *import different WASI modules (B.2)* to support specific functionalities. These imports encounter a few bugs, such as module not found, incompatible version, etc. These bugs account for 5.6% in WASI Robustness. Due to the updated versions of runtimes, new WASI versions are continuously provided by the runtimes. Using the *incompatible WASI version (B.6)* in WASM runtimes could be unsupported and account for 7.4% of bugs in WASI Robustness.

Moreover, WASI is the bridge between WASM and the OS, which should support different OSs. The diversity in these OSs can result in *Operating system support error (B.5)*, accounting for 5.6% of bugs in this category.

Furthermore, all the WASM runtimes should support WASI to interact with the low-level system, and `wasmer` also provides another application binary interface (ABI) to do this. Bugs in this part are regarded as *Other counterpart error (B.7)* that account for 9.3% of bugs in this category.

4.3 Runtime Environment

After compiling WASM to native code, WASM runtimes support the WASM with an execution environment. The runtime environment supports WASM with module import, trap message tracking, metering computing cost, and other functionalities. Bugs happening in the *Runtime Environment (C)* account for 38.6% of bugs in total.

We observe a significant proportion (20%) of bugs about module operation in Runtime Environment, including *Module instantiation bugs (C.1)* and *Module import error (C.2)*. Specifically, 13.3% of bugs in this category are related to WASM module instantiation. WASM module has to be instantiated before execution. These bugs are related to the instance allocator, module loading, multiple instantiation error, etc. High-level language API makes it easier for WASM developers to utilize WASM runtimes. The API could import modules from the host environment or other WASM modules. These bugs are about unknown imports, calling host functions, etc.

Functions in WASM could *call the host functions defined in high-level language (C.3)* and account for 10% bugs in *Runtime Environment (C)*. This process contains bugs of parameter passing, finding host functions, etc.

Memory issue (C.4) is a common kind of bugs when executing the WASM binaries, accounting for 15.8% bugs in this category. These bugs are about memory management, including memory allocation, multi-memory support, out-of-memory error, memory release and memory growth.

When executing WASM binaries, WASM runtime could encounter bugs in dealing with *traps (C.5)* and lead to an abortion, accounting for a total of 7.5% of bugs in this category. These bugs are related to the process of the `unreachable` instructions in WASM binaries. Besides, WASM runtimes sometimes do nothing with the errors, and the errors were not carefully reported to users. The WASM runtime should *generate an exception or return a well-defined error (C.10)*.

In executing native code generated from WASM, many users encounter *Thread safety issue (C.7)* and *Stack issue (C.8)*. *Thread safety issue (C.7)* refers to the thread safety when executing WASM binaries. *Stack issue (C.8)* refers to the bugs about the stack, such as match rules for popping when calling WASM functions. These bugs account for a total of 13.3% of bugs in *Runtime Environment (C)*.

We also observe three *bugs about the entry point of a WASM module (C.9)*. The functions named "_main", "_start", "main", and "start" are regarded as entry points of a WASM module. A WASM runtime will call the entry point function by default without setting the function name through the command line or high-level language API. However, some WASM runtimes require each WASM module to hold an entry point which is too strict. These bugs are summarized as *Entry point error (C.9)*.

Furthermore, when developers use high-level language API to do some operations of a WASM module, they may encounter *data type conversion problems (C.11)*, accounting for 3.3% of bugs in the current inner bug category.

Besides, the Runtime Environment can not meet all expectations of functionalities from users. Runtime Environment *lacks support for some features (C.6)* that users need, which account for 3.3% of bugs on its own.

4.4 Auxiliary tools

Besides executing WASM files, WASM runtimes provide users with handy little tools related to WASM, including validating the format of WASM files, WASM module cache, Wat and WASM

Table 2. Statistics of the four additional runtimes.

Runtime	Stars	Commits	Language	GitHub issues
wasm3	6k	1661	C/C++	14 (8)
WasmEdge	5.8k	2562	C/C++	83 (42)
wasmer-python	1.8k	1024	Rust & Python	33(6)
wasmer-go	2.4k	668	Go & C	54(14)
Total			**184 (70)**	

*The refined numbers are in the parentheses.

files conversion and package manager. As different WASM runtimes differ significantly in this respect, this category is not classified into leaf categories. This category accounts for 5.8% of all the classified bugs. For example, in `wasmer` issue #2028 [8], when passing environment variables into `wasmer` run via the *–env* flag, the program will fail if the environment variable contains an '=,' which should be allowed. Moreover, when validating the format of WASM binaries, `wasmer` uses command *wasmer validate* to do this. Although setting the parameter *–enable-simd*, it incorrectly reports an error when validating a WASM module with SIMD.

4.5 Validity of the Taxonomy

Three WASM runtimes (`wasmer`[33], `wasmtime`[36], `WAMR`[40]) are considered to construct the bug taxonomy. These three WASM runtimes are only implemented in Rust and C++. However, it is unknown whether the bug taxonomy is generalizable. To address this concern, we need to assess the taxonomy on other WASM runtimes to ensure its generalizability. Therefore, we include WASM runtimes implemented in different languages, including `wasm3`[26], `WasmEdge`[32], `wasmer-python`[35] and `wasmer-go`[34], to consider as diverse WASM runtimes as possible. We extract closed Github issues with the bug label of the other four WASM runtimes (`wasm3`[26], `WasmEdge`[32], `wasmer-python`[35] and `wasmer-go`[34]) and classify them according to the proposed taxonomy. First, we filtered out issues with no definite answers to ensure the accuracy and certainty of bugs and fix strategies. Second, we exclude installation/build bugs, documentation bugs, and other issues unrelated to WASM binaries' execution from the source data. Finally, as shown in Table 2, the total number of the four WASM runtime bugs is 70.

As all the bugs in the four WASM runtimes can be classified according to the taxonomy mentioned above, the threat could be considered neutralized. Although WASM runtimes have their characteristics, they show a similar architecture, as shown in Figure 3. The bug categories of all the bugs from the four runtimes (`wasm3`, `WasmEdge`, `wasmer-go` and `wasmer-python`) are also published in the artifact [44].

Summary of answers to RQ1:
(1) *We construct a taxonomy of 31 leaf bug symptom categories in WASM runtimes, indicating the root causes and the diversity.*
(2) *Bugs in Backend Compilation account for 38.3% of WASM runtime bugs, covering 10 leaf categories. A large proportion (23.5%) of these bugs are revealed with incorrect compilation.*
(3) 17.4% of bugs are related to WASI implementation, covering 9 symptom categories. In particular, 46.3% of the bugs in this category are related to the basic functionalities of WASI (i.e., B.1, B.2, and B.4).
(4) Most (i.e., 38.6%) bugs occur in the Runtime Environment, covering a broad spectrum of symptoms (i.e., 12 leaf categories). Among them, memory issue is the most common, accounting for 15.8% of bugs in this category.

5 RQ2:FIX STRATEGIES OF WASM RUNTIME BUGS

To figure out how developers fix various types of bugs, we distill their fix strategies in this section for each inner bug category. Due to bugs in categories *Auxiliary tools* are either too specific or irrelevant to WASM runtime themselves, and they only account for 5.8% of bugs, we do not study the fix strategies for them. We have summarized the general fix strategies for the remaining three inner symptom categories. As shown in Figure 8, 11 and 12, the X axis shows each leaf bug category in Figure 5, and the Y axis represents the corresponding fix strategies following with their totally used frequency under the inner category. We elaborate on the summarized fix strategies of their frequent symptoms and demonstrate some examples of bugs and corresponding fixes in the real world.

5.1 Fix Strategies for Backend Compilation

We summarize eight systematic fix strategies for bugs in Backend Compilation and illustrate the distribution of these strategies on leaf categories in Figure 8.

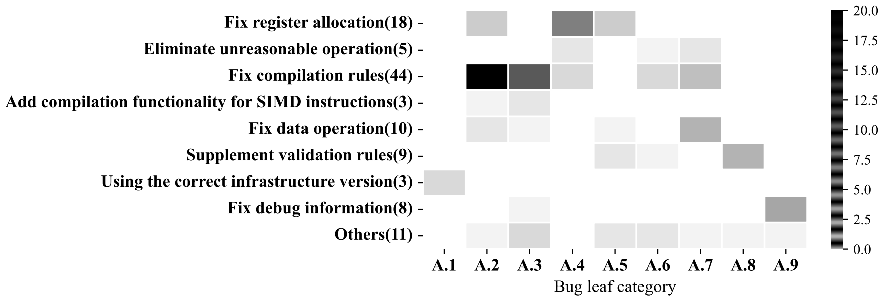


Fig. 8. Distribution of fix strategies for Backend Compilation.

Fix compilation rules. 39.6% of bugs in Backend Compilation can be solved by modifying compilation rules in different backend compilers. Compilation rules are the guide for translating WASM instructions to native code. For example, `wasmtime` developers modify the files with `.isle` suffix to change the rules of emitting native code. As shown in Example (b) (Figure 7), during the lowering of SIMD instructions, the allocation of registers in `wasmtime` shows a bug [15]. The `movdqa` instruction moves out of `v6`, but `v6` is never set. This bug is because the SIMD lowering is missing an instruction somewhere. To fix this bug, the `wasmtime` contributors modify the `lower.isle` file to complement the compilation rules for the *i64x2*, *i32x4* and *i16x8* extend instructions. `Wasmer` fixes the emitter file for different CPU architectures to emit native code. This fix strategy covers five bug symptoms and is especially frequently adopted in the *Incorrect compilation (A.2)* and *Compilation failure (A.3)* bug categories. 71.4% of *Incorrect compilation (A.2)* bugs are fixed after modifying the compilation rules in the backend compilers to support more reasonable translation. As for *Compilation failure (A.3)* bugs, the backend compilers may encounter unexpected exceptions and abortion due to the unreasonable compilation rules. Therefore, developers fix these bugs by changing the compilation rules to meet the actual requirements and support emitting correct native code in most cases.

As shown in Example (c) (Figure 9), a developer reports that the *i64.rotr* instruction in WASM is incorrectly compiled with LLVM in `wasmer` when given a rotate amount of 0 (line 5 with highlight) [10].The *i64.rotr* instruction takes two operands: the first operand is the value to be rotated (a 64-bit integer), and the second operand specifies the number of bits to rotate. The instruction takes the bits from the input value and shifts them to the right by the specified number of bits. The bits that are shifted out from the right side reappear on the left side, creating a circular rotation.Rotate by 0 should not change the first operand, and the expected result is 4. However, `wasmer` mistakenly compiles the *i64.rotr* instruction with amount 0 and get the result of -1. The contributors modified the lowering rules for shift operations when translating from WASM instructions to LLVM IR. As depicted in Figure 9, v1 represents the value to be rotated, and v2 is the number of bits to rotate. The code uses a bitwise AND operation with a bit mask to ensure that the shift amount includes 0. Specifically, the modified program creates a 64-bit integer constant with all bits set to 1, except for the most significant bit (bit 63), which is set to 0. This mask ensures that the shift value falls within the valid range. Subsequently, a bitwise AND operation is performed between v2 and the mask to ensure the shift value v2 is between 0 and 63. The program then calculates the left and right parts of the rotation using left and right bitwise shift operations, respectively. Furthermore, the program combines these parts using the bitwise OR operation to obtain the final result. Finally, the result is pushed back onto the stack. Compilation rules, such as lowering rules, provide guidance for translating WASM instructions into native code.

Even worse than emitting incorrect native code is that the backend compilers fail to compile some instructions or the whole WASM module. For example (`wasmtime` issue #2347 [5]), a user reports that the backend compiler in `wasmtime` (V0.20 and main branch) fails to compile a WASM module. The `wasmtime` developers fix it by modifying the compilation rules. In detail, they do block manipulation in the `wasmtime` translation of some table-related instructions and explicitly call the *ensure_inserted_block()*.

Fix register allocation. 16.2% of bugs in Backend Compilation, involving three frequent bug categories, can be fixed by changing the register allocation. As aforementioned in Section 4, while generating native code from WASM instructions, the backend compilers are expected to allocate registers. Nevertheless, they may use incorrect registers, load data from an unexpected register, exhaust registers, etc. Various WASM instructions and instruction set architectures (ISA) make it hard for the backend compilers to allocate suitable registers.

Fix data operation. The fix strategy is used for 9.0% of bugs in Backend Compilation, covering a wide range of bug categories, including *Incorrect compilation (A.2)*, *Compilation failure(A.3)*, *Incomplete operating system support (A.5)* and *Unsupported data operation (A.7)*. The strategies include fixing data alignment, adding support for i8, i16, fixing byte order, dealing with undefined upper bits, converting data types, returning multi-value data, supporting v128 data type, etc.

Supplement validation rules. Supplement rules of verifier will tackle the problems in validation, repairing 8.1% of bugs in Backend Compilation, and mainly fix the *Validation error (A.8)*. For example (`wasmer` issues #2187 [9]), a developer reports that he can not get the memory page in WASM of

Fault description : The *i64.rotr* instruction is mis-compiled with LLVM backend compiler in wasmer when given a rotate amount of 0.
Fault symptom : [A.2] Incorrect compilation

Fix strategy : Fix compilation rules

wat code :

```
(module
 (func () (result i64)
   i64.const 4
   i64.const 0
   i64.rotr)
 (export "_main" (func 0)))
```

modified program :

```
let v1 = self.apply_pending_canonicalization(v1, i1);
let v2 = self.apply_pending_canonicalization(v2, i2);
let (v1, v2) = (v1.into_int_value(), v2.into_int_value());
let mask = self.intrinsics.i32_ty.const_int(31u64, false);
let v2 = self.builder.build_and(v2, mask, "");
let lhs = self.builder.build_left_shift(v1, v2, "");
let rhs = {
   let negv2 = self.builder.build_int_neg(v2, "");
   let rhs = self.builder.build_and(negv2, mask, "");
   self.builder.build_right_shift(v1, rhs, false, "")
};
let res = self.builder.build_or(lhs, rhs, "");
```

Fig. 9. Example (c) - GitHub wasmer issue #2215

65536, although the user sets memory minimum and maximum sizes range 0..65536 inclusive by WASM instructions. The verifier is expected to block 65537 and higher. However, wasmer only works on 65535 and lower. This corresponding fix strategy is to modify its validation rules.

Fix debug information. This fix strategy repairs 7.2% of bugs in *Backend Compilation (A)*, dealing with 87.5% bugs in *WASM debugging information error (A.9).WASM debugging information error (A.9)* may lead to the failure of providing incorrect debug information that misleads the users. By fixing debug information, these bugs could be well settled.

Eliminate unreasonable operation. Some functionalities in backend compilers of WASM runtimes lead to an unexpected consequence. These functionalities are meaningless and need to be limited. For example (wasmtime issue #2883 [13]), wasmtime users try to use ssub_sat with two I64 values. They use cranelift-object with a triple in the intermediate presentation (IR) in the wasmtime backend compiler. It is worth mentioning that ssub_sat is a vector command, but it is used with a scalar. Moreover, the developers eliminate the unreasonable operation, limiting saturating arithmetic instructions: uadd_sat, sadd_sat, usub_sat, and ssub_sat, and applying them only to vector types. This kind of fix strategy fixes 4.5% of bugs in Backend Compilation.

Fault description : The compilation panic when lowering WASM *select* instruction with v128 inputs.

Fault symptom : [A.3] Compilation failure

Fix strategy : Add compilation functionality for SIMD instructions

wat code :

```
(module
  (func (result v128)
    v128.const i32x4 0x00000000 0x00000000 0x00000000 0x00000000
    v128.const i32x4 0x00000000 0x00000000 0x00000000 0x00000000
    i32.const 0
    select))
```

Fig. 10. Example (d) - GitHub wasmtime issue #3173

Add compilation functionality for SIMD instructions. Some WASM runtimes do not support the intact functionalities to deal with SIMD instructions. Adding the support will address the bugs. This strategy fixes 3 bugs in *Incorrect compilation (A.2)* or *Compilation failure (A.3)* related to SIMD instructions. The *select* instruction allows for the conditional selection of one of its first two operands, depending on whether its third operand is zero. When the third operand is zero, the second operand is selected. Otherwise, the first operand is selected.However, as shown in Example (d) (Figure 10), the *select* instruction in `wasmtime` fails an assertion when it is given v128 types as operands (lines 3,4,6 with highlight) [14]. It is due to the vector types are not supported for the *select* instruction. The contributors fix it by adding the vector types check for SIMD instructions.

Using the correct version of the infrastructure. Since some backend compilers in WASM runtimes rely on existing frameworks such as LLVM, adjusting the LLVM's version can handle some problems. This fix strategy fixes all the bugs in *Incompatible infrastructure version (A.1).*

5.2 Fix Strategies for WASI Robustness

As illustrated in Figure 11, we identify seven frequent fix strategies for bugs in WASI Robustness.

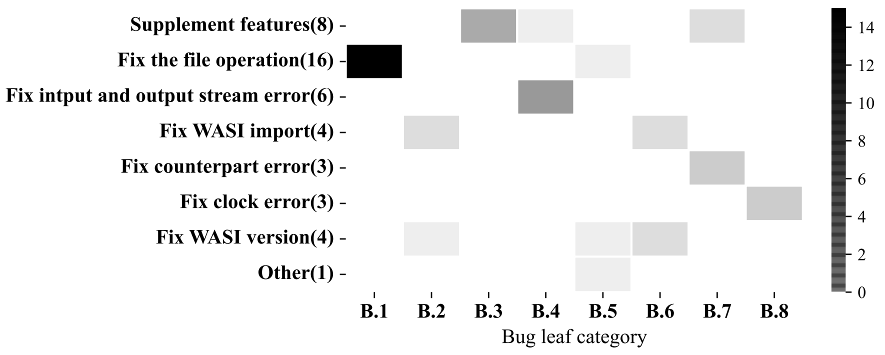


Fig. 11. Distribution of fix strategies for WASI Robustness.

Fix the file operation. This strategy fixes 35.6% bugs in WASI Robustness, including all the bugs in *File operation error (B.1)* and half of the bugs in *Operating system support error (B.5)*. For example (`wasmer` issue #2297 [11]), a user reports that renaming a temporary file through a WASM file fails and prints `unable to rename temporary`. The developers fix the implementation of WASI to allow this operation.

Supplement features. WASM runtimes are expected to implement the necessary features. However, some WASM runtimes do not fulfill this expectation. In such cases, WASM runtime developers need to implement the expected functionalities in WASI and thus the *Unsupported operation (B.3)* bugs can be fixed.

Fix input and output stream error. When the required message is not successfully printed, or the necessary information is not correctly imported into WASM, the *Input and output stream error (B.4)* occurs. In these cases, developers need to fix the input and output streams. This fix strategy fixes 13.3% bugs in WASI Robustness.

Fix WASI import & Fix the WASI version. The two strategies are mainly used to tackle the *Import error (B.2)* and *WASI version error (B.6)* which occur when using different modules from WASI. The developers fix WASI import to use the suitable WASI module to support specific functionalities, and fix WASI version to make the WASM binaries compatible with the current circumstances. The two strategies all fix 8.9% of bugs in WASI Robustness.

Fix counterpart error. This fix strategy can resolve the *Other counterpart error (B.7)*, accounting for 6.7% of bugs in WASI Robustness, which could be regarded as the repair method for all the counterparts ABI.

Fix clock error. The fix strategy only focuses on the *Clock bugs (B.8)* in WASI Robustness and resolve 6.7% bugs.

5.3 Fix Strategies for Runtime Environment.

We identify eleven frequent fix strategies for bugs in Runtime Environment and Figure 12 shows the distribution.

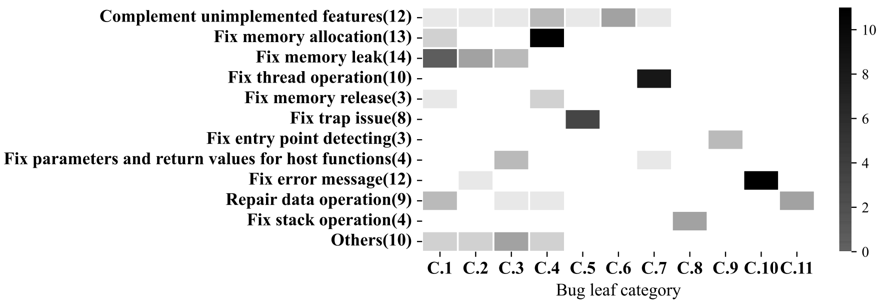


Fig. 12. Distribution of fix strategies for Runtime Environment.

Fix memory allocation & Fix memory leak & Fix memory release. 29.4% of bugs in Runtime Environment can be resolved by the three fix strategies for memory management. The three strategies mainly fix the *Module instantiation bugs (C.1)* and *Memory issues (C.4)*. After compiling the WASM binaries into native code, WASM runtimes need to instantiate the current WASM module. Errors about instance allocation and module loading errors could happen. Besides, other bugs related to multi-memory support, out-of-memory, and memory growth could occur. The developers mainly use these methods to deal with such cases.

Fix error message. This fix strategy is to modify the error message to make it more reasonable or catch unexpected failures with an error message. This strategy fixes 11.8% bugs in Runtime Environment. For instance, a user reports (`wasmer` issue #830) that the WASM runtime should throw an error instead of panicking when a string is given instead of a digit in the WASM program [2]. This issue needs to be addressed by modifying the error message. In response, a contributor has added further details to the error message: "Cannot parse the provided argument as an integer."

Complement unimplemented features. The fix strategy resolves a wide range of bugs in the Runtime Environment, including Module instantiation bugs (C.1), Module import error (C.2), Calling host functions (C.3), Memory issue (C.4), Trap error (C.5), Unsupported features (C.6) and Thread safety issue (C.7), accounting for 11.8% of the total number.

Fix thread operation. 83.3% of the bugs in the Thread safety issue (C.7) are fixed by this resolution. For example (`WAMR` issue #1144 [3]), a user reports that the *wasm_runtime_atomic_wait* is not thread-safe and one of subthreads call *wasm_runtime_spawn_exec_env* return nullptr. The developers use this strategy to fix atomic wait to be thread-safe by a lock.

Fix trap issue. This strategy is entirely used to fix trap error (C.5), accounting for 7.8% of bugs in Runtime Environment, including fixing trap catch, complementing the missing trap information, and so on.

Repair data operation. 8.9% of bugs in Runtime Environment are fixed by repairing data operation, which mainly address the data type mismatch between WASM and high-level language API. Besides, it also fixes data alignment. All the Data type conversion (C.11) bugs are fixed by repairing data operation.

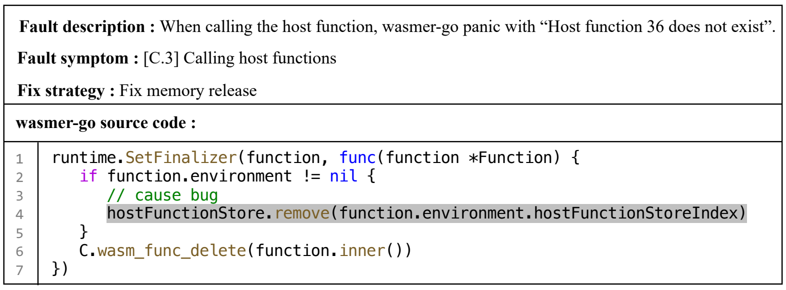
Fix parameters and return values for host functions. As suggested in Figure 12, 25% of bugs in Calling host functions (C.3) are fixed by modifying the parameters and return values because there is a mismatch or passing error between the parameters and return values.

Fix entry point detecting. 2.9% of bugs in Runtime Environment are resolved by fixing the detection of the default entry point. The WASM runtime needs to check the existence of the specially named entry function before execution.

Fix stack operation. All the bugs in *Stack issues (C.8)* are resolved by this strategy. The WASM runtime is expected to fix the order of the items when popped from the stack to match the WASM invocation rules.

5.4 Validity of the Fix strategies

The three WASM runtimes (wasmer, wasmtime, and WAMR) are studied to characterize fix strategies of bugs. The threat identified in the bug taxonomy mentioned in Section 4 also exists in the fix strategy. It also needs to be classified whether the fix strategies can be generalized to other WASM runtimes. To validate the fix strategies, we also summarize the fix strategies of issues extracted from the four runtimes (wasm3, WasmEdge, wasmer-go and wasmer-python), as illustrated in Section 4.5.


Fault description : When calling the host function, wasmer-go panic with “Host function 36 does not exist”.

Fault symptom : [C.3] Calling host functions

Fix strategy : Fix memory release

wasmer-go source code :

```
1 runtime.SetFinalizer(function, func(function *Function) {
2    if function.environment != nil {
3       // cause bug
4       hostFunctionStore.remove(function.environment.hostFunctionStoreIndex)
5    }
6    C.wasm_func_delete(function.inner())
7 })
```

Fig. 13. Example (e) - GitHub wasmer-go issue #244

The summarized bug fixing strategies are helpful for all the WASM runtimes as the fix strategies in these four WASM runtimes could be classified into the existing fix strategies. The fix strategies of all the bugs from the four runtimes (wasm3, WasmEdge, wasmer-go and wasmer-python) are also published in the artifact. For example, when the user tries to execute an exported WASM function, wasmer-go would panic with "Host function 36 does not exist" (the index of the function would vary as well). However, the user indeed registered more than 36 imported functions [6]. It is due to the function finalizer shown in Example (e) (Figure 13). While executing, all the *hostFunctionStore* functions are set to nil due to the early garbage collection in line 4, causing panic. This bug in wasmer-go is classified to *C.3 Calling host functions*. The contributors are adding the imports to the Instance struct to prevent this premature garbage collection from happening. The fix strategy of this bug is classified to Fix memory release.

As all the bugs in the four WASM runtimes can be classified according to the taxonomy and the summarized bug fixing strategies are helpful for all the WASM runtimes, the threat could be considered neutralized.

Summary of answers to RQ2:
(1) We identify eight systematic fix strategies for bugs in Backend Compilation. The three most common strategies are fixing compilation rules, registering allocation, and data operation, resolving 39.6%, 16.2%, and 9.0% of bugs in this category, respectively.
(2) We distill seven systematic strategies for bugs in WASI Robustness. The most common one is fixing the file operation, which resolves 35.6% of bugs in this category.
(3) The fix strategies for bugs in Runtime Environment are diverse, including fixing memory leaks, fixing memory allocation, complementing unimplemented features, etc. Moreover, the most commonly used fix strategies in Runtime Environment are fixing memory allocation and fixing error messages.

6 RQ3:PATTERN-BASED BUG DETECTOR FOR WASM RUNTIMES

Our aforementioned analysis suggests that most bugs have specific patterns and share similarities across different WASM runtimes. Thus, in this section, we seek to develop a pattern-based bug detection framework to identify bugs in WASM runtimes. Our key idea is to construct test cases that can trigger various kinds of bugs we summarized in Section 4. Specifically, we seek to construct one or more test cases to trigger each of the summarized bug category. Note that, the constructed test cases are either re-constructed from the bug reports or we create them from scratch according to the bug patterns.

Next, we will present the details of our bug-triggering test cases for the three major bug categories: *Backend compilation (A)*, *WASI Robustness (B)* and *Runtime envrionment (C)*. Note that, *Auxiliary Tools (D)* is an additional part provided by WASM runtimes, and the tools provided by WASM runtimes in this part vary considerably. Thus, the category *Auxiliary Tools (D)* is not considered in this section. Further, for some leaf categories, it is hard for us to construct test cases, which thus are not covered by our detection framework. In total, our detection framework is constituted of the following 19 bug detectors.

6.1 Bug detectors for Backend Compilation

[A.2] Incorrect compilation. SIMD instructions are the newly introduced features for WASM binaries. WASM runtimes show the bug pattern when compiling specific SIMD instructions, such as using `i64x2` and `i32x4` to simulate the v128 type and the optimization for them. To identify such bugs, we select typical WASM binaries with these instructions to do the detection.

[A.3] Compilation failure. We develop the bug detector to identify two typical failures: a compilation error that occurs when handling instructions with v128 (128-bit length vectors) as parameters and a failure during the compilation of large WASM modules. The bug detector includes a WASM program with the "select" instruction using v128 as the parameter. Additionally, we utilize an existing large WASM module named "Ti database", which incorporates all its backend compilers for WASM runtimes. Even worse, WASM runtimes could fail to generate native code for some instructions, especially those with v128 as parameters or the large WASM modules. We construct the bug detector to detect the *select* instruction with v128 as the parameters. Moreover, we use a large WASM module Ti database with all the backend compilers in WASM runtimes.

[A.4] Register allocation error. WASM runtimes could load data from an undefined register or get fused with other instructions when compiling the specific instructions, such as *i64x2.extend_low_i32x4_u* and *f64x2.replace_lane*. To identify such bugs, we select typical WASM binaries with these instructions to do the detection.

We extract the specific OS-related bug-triggering WASM modules for *[A.5] Incomplete operating system support* and unsupported data operation such as alignment of SIMD for *[A.7] Unsupported data operation.*

We use the max value linear memory to detect the *[A.8] Validation error* and WASM binaries from bug reports which easily trigger debugging information to detect the *[A.9] WASM debugging information error.*

6.2 Bug detectors for WASI Robustness

[B.1] File operation error. Different WASM runtimes show similar bug patterns about file operation errors. These easily bug-triggering file operations include renaming, moving, counting, and mapping. Thus, based on the shared file operation bug types mentioned in the bug issues, we design the bug detector to detect these bug types. For example, we test whether WASM runtimes could rename a file or report error information when the file does not exist. Besides, the detector could test whether WASM runtimes can move a file, count the file number in a directory, or do a mapping operation.

[B.2] Import error. The most commonly found bug about import is that some WASM runtimes could not support importing multiple WASI versions in one WASM module. Thus, we import both *wasi_snapshot_preview1* and *wasi_unstable*, the most used WASI versions, in one WASM module to detect this bug.

We extract the WASM binaries, including the unsupported pre-opened directories with / and ./ for WASI, to detect the bug in *[B.3] Unsupported operation.*

[B.4] Input and output stream error. To support detecting bugs about standard input and output stream, we use C++ and compile the C++ program into WASM binaries by emscripten[43] to see the rights and types in *__wasi_filestat_t*. If the WASM runtime could not successfully print the expected result, it could be a bug. For example,`wasmtime` prints OS error when detecting this kind of bug, which the developer confirms.

[B.5] Operating system support error. There are two OS-specific parts of WASI implementation: clocks and polling. Due to the difference among OSs, the same operation could fail in a specific OS. For example, the QuickJS engine based on WASM binaries only fails in Windows due to the differences between POSIX and Windows async APIs. We extract the QuickJS engine from bug issue to detect this bug.

6.3 Bug detectors for Runtime environment

[C.1] Module instantiation faults. WASM runtimes provide various high-level language APIs for users to execute WASM binaries embedded in different applications. When running WASM binaries in a high-level language, the first step is to load the WASM module from a file or directly load the textual format WASM module in a string variable. And then, instantiate the WASM module, including validating the WASM module, compiling the WASM binaries with the appointed backend compiler, allocating the memory allocation for the table, global, etc. However, WASM runtimes could not support the instantiation for an empty module. Some WASM runtimes will encounter memory leaks when instantiating multiple WASM modules in a short time. Thus, we use the bug detector to detect whether WASM runtimes support instantiating an empty WASM module and whether it will lead to memory leaks when instantiating multiple WASM modules in a short period.

[C.2] Module import error. We observe that some WASM runtimes omit the step to check the index of imported items, such as skipping to report the error of *index out of bounds* errors when *import_global_index* is greater than imports. globals length. We extract the related WASM binaries from the raw bug report to detect this bug by the bug detector.

[C.3] Calling host functions. We use the bugs detector for this kind of bug to detect whether WASM runtimes could support importing a self-defined module, not only from *env.* Besides, some

WASM runtimes show the bug pattern about mis-mapping multiple host functions. We use the bug detector to test whether WASM runtimes could successfully run the functions by importing them in the correct order or if the runtime could inspect the mapping by importing them in the wrong order and report the error message.

[C.4] Memory issue. By the bug detector, we detect whether WASM runtimes could grow the linear memory dynamically. We extract the WASM module from bug issues and modify it to grow the memory using *memory.grow* instruction and using *memory.size* instruction to check the linear memory size after the growth.

[C.5] Trap error. These bugs are related to the process of the *unreachable* instructions in WASM modules. By the bug detector, we use a WASM module with *unreachable* instructions to test whether WASM runtimes could successfully break the execution and report the information in the location where *unreachable* is.

[C.9] Entry point error. WASM runtimes are expected to regard the function labeled with 'start' or '_start' as the entry point and execute this function default and allow the WASM module without an entry point. WASM runtimes show a similar bug pattern about the entry point: do not run the entry point function or reject the WASM modules without an entry point. We construct the WASM module without or with an entry point to detect this kind of bug.

[C.10] Unhandled error. Some WASM runtimes usually encounter panic directly without any operation to avoid it by reporting the error information. The most commonly found are unhandled errors with unsupported operation and invalid access to the data section. We extract typical WASM module examples to detect this bug.

6.4 Reliability of the bug detectors

As a portion of the bug detectors are curated by ourselves based on the code snippets and bug description provided in the bug reports, we first need to evaluate the reliability of the constructed bug detector. Note that, we already have the ground truth, i.e., WASM runtimes (with specific version) have some kinds of bugs. Thus, we make effort to reconstruct the environment (i.e., OS, WASM runtime version, configuration, etc.) to replicate the reported bug for each category. At last, the bug detector can trigger the reported bugs, which suggest the reliability of our detection framework.

6.5 Detecting new bugs

As the bug detector we create was constructed based on the knowledge summarized from `wasmer`, `wasmtime`, and `WAMR`, we further apply it to different WASM runtimes, seeking to identify new bugs.

Experimental Setting. In this experiment, we consider the seven mentioned WASM runtimes, i.e., `wasmer`, `wasmtime`, `WAMR`, `wasm3`, `WASMEdge`, `wasmer-python` and `wasmer-go` to investigate the generalizability of our study. The bug detector is applied to the following WASM runtimes: `wasmer 2.3.0`, `wasmtime 0.38.0`, `WAMR 05-18-2022`, `wasm3 0.5.0`, `WASMEdge 0.9.1`, `wasmer-python 1.1.0` and `wasmer-go 1.0.4` on different execution modes (interpreter, AoT, JIT) and across three different operating systems (`macOS 10.15`, `Ubuntu 20.04`, and `Windows 11`). However, `wasmer-python` and `wasmer-go` do not support `Windows 11`. Thus the evaluation of `wasmer-python` and `wasmer-go` are excluded.

Result. As shown in Table 3, we find 60 new bugs, covering all the tested WASM runtimes. By the time of this submission, 13 of them have been confirmed by the developers, with 9 already been fixed in the main branch based on our suggestions. Moreover, the number of test cases and the number of bugs found, confirmed, and fixed in leaf categories are shown in Table 4. To be more rigorous, we only include the zero-day bugs that were first revealed by us. We also find some bugs that have been reported by others, and these bugs are excluded from our statistical data.

Table 3. The experiment result of the bug detector. We mark a leaf category on a WASM runtime as 0 if it passes all the execution modes across all the OS platforms. Otherwise, it is marked with the number of detected bugs.

Leaf category	**wasmer**	**wasmtime**	**WAMR**	**wasm3**	**WasmEdge**	**Wasmer-python**	**Wasmer-go**
[A.2]	0	0	3	1	2	1	1
[A.3]	0	1	1	2	3	0	0
[A.4]	0	0	1	0	0	2	1
[A.5]	0	0	1	0	1	1	0
[A.7]	0	0	1	0	1	0	0
[A.8]	0	0	1	1	0	0	1
[A.9]	1	0	1	0	2	0	0
[B.1]	0	0	3	4	2	0	0
[B.2]	0	0	0	0	1	1	1
[B.3]	0	0	0	1	1	0	0
[B.4]	0	0	1	1	1	0	0
[B.5]	0	0	0	0	0	0	0
[C.1]	0	0	0	0	3	0	0
[C.2]	0	0	0	1	0	1	0
[C.3]	0	0	0	0	1	0	0
[C.4]	0	0	1	0	1	0	0
[C.5]	0	0	0	1	0	0	0
[C.9]	0	0	0	0	0	0	0
[C.10]	0	0	0	2	1	0	0
Bugs found	1	1	14	14	20	6	4
Total	60						

Test target: This test case tests whether a WASM runtime could correctly compile the *rotr* instruction in WASM binaries.

wat code :

```
(module
  (func () (result i64)
    i64.const 4
    i64.const 0
    i64.rotr)
  (export "_main" (func 0)))
```

Expected result : 4

Fig. 14. Example of bug found in `[A.3] Incorrect compilation`

Examples of bugs found. As shown in Figure 14, it is expected to print the number 4 when testing the `rotr` instruction for WASM binaries. However, the actual output in `WAMR` is a random number. Every time executing, it leads to a different output. The developers have confirmed it is a bug and fixed it in the main branch [21], dealing with the parameter 0 separately. This bug belongs to *Incorrect compilation (A.3)*.

An additional example is shown in Figure 15. It is expected to print the correct directory number 203 when testing `WASI` in the runtime. However, `WasmEdge` prints 147 as a result, which is already confirmed as a new bug by the developers [31]. Once the number of files is larger than 147, it will be truncated in `WasmEdge`. And the file renaming belongs to *[B.1] File operation error* fails in `wasm3`, which is also confirmed and fixed [29].

Table 4. The test case number and the bug number of found, confirmed and fixed.

Leaf category	**#Test case**	**#Bug found**	**#Bug confirmed**	**#Bug fixed**
[A.2]Incorrect compilation	7	8	2	2
[A.3]Compilation failure	5	7	3	2
[A.4]Register allocation error	3	4	0	0
[A.5]Incomplete operating system support	2	3	0	0
[A.7]Unsupported data operation	1	2	1	0
[A.8]Validation error	1	3	2	2
[A.9]WASM debugging information error	2	4	2	1
[B.1]File operation error	6	9	2	1
[B.2]Import error	1	3	0	0
[B.3]Unsupported operation	1	2	0	0
[B.4]Input and output stream error	1	3	0	0
[B.5]Operating system support error	1	0	0	0
[C.1]Module instantiation bugs	3	3	1	1
[C.2]Module import error	1	2	0	0
[C.3]Calling host functions	1	1	0	0
[C.4]Memory issue	1	2	0	0
[C.5]Trap error	1	1	0	0
[C.9]Entry point error	1	0	0	0
[C.10]Unhandled error	3	3	0	0
Total	42	60	13	9

Test target: This test case tests the read dir functions for WASI.
WASM file code : The WASM file is more than 2000 lines and is limited to show here. It is provided in the artifact.
Expected result : 203

Fig. 15. Example of bug found in `[B.1] File operation error`

As shown in Figure 16, it is expected to allocate the linear memory to the max value. However, the allocation fails in `WAMR` and `wasm3`. This bug belongs to *Validation error (A.8)* since the max value is not permitted by the validator. The developers in `WAMR` updated the max memory page value in the interpreter [22], and the developers from `wasm3` updated the max linear memory pages from 32768 to 65535 [27].

As shown in Figure 17, it is expected to print the value 340282366920938463463374607431768211455 when executing the *func1* function in the WASM binaries. However, `wasmer-go` print the value 0. `Wasmer-go` could not correctly compile the *f32x4.abs* and *v128.not* instructions in the WASM binaries.

Interestingly, we find that the test cases in our detection framework can trigger more than one types of bugs. For Example, the WASM module in Figure 18 is used to test whether WASM runtimes could successfully compile the *div* and *copysign* instructions for float data (*[A.3] Compilation failure*). Beyond this, we found that it can identify bugs that belong to *[C.9] Entry point error* in `wasm3`. The

Test target: When allocating WASM linear memory with the max value, memory allocation failed. This is rejected by the validator in the backend compiler.

wat code :

```
(module
  (memory (;0;) 65536)
)
```

Expected result : Successfully allocate WASM linear memory.

Fig. 16. Example of bug found in `[A.8] Validation error`

WASM module could be successfully compiled in `wasm3`. However, '_start' is not considered the entry point in `wasm3`, although in other runtimes it is. The developer confirmed it and considered fixing it by checking the return type of '_start' and acting according to it [28]. Moreover, the WASM module in Figure 19 is used to test whether WASM runtimes could successfully compile the *select* instruction with two v128 parameters (*[A.3] Compilation failure*). It detects a bug in `wasm3` which should be summarized to *[A.7] Unsupported data operation.* Although `WasmEdge` could compile the module, it does not support printing v128 data. The developer confirmed that they only support print i32, i64, f32, and f64, which posed a bug, and they fixed it [30].

Test target : To test whether a WASM runtime could successfully compile the *f32x4.abs* and *v128.not* instructions in the WASM binaries.

wat code :

```
(module
  (func (result v128)
    v128.const f32x4 0 0 0 0
    f32x4.abs
    v128.not)
  (export "func1" (func 0))
)
```

Expected result : 340282366920938463463374607431768211455

Fig. 17. Example of bug found in `[A.2] Incorrect compilation`

Test target: This test case tests whether a WASM runtime could correctly compile the *div* instruction and *copysign* instruction for float type and execute the start function by default.

wat code :

```
(module
  (type (;0;) (func (result f64)))
  (func (;0;) (type 0) (result f64)
    f64.const 0x0p+0 (;=0;)
    f64.const 0x0p+0 (;=0;)
    f64.const 0x0p+0 (;=0;)
    f64.div
    f64.copysign)
  (export "_start" (func 0)))
```

Expected result : 0

Fig. 18. Example of bug found in `[C.9] Entry point error`

Test target: This test case tests whether a WASM runtime could successfully compile the *select* instruction with two v128 parameters and execute the start function by default.

wat code :

```
(module
  (func (result v128)
    v128.const i32x4 0x00000009 0x00000000 0x00000000 0x00000000
    v128.const i32x4 0x00000009 0x00000000 0x00000000 0x00000000
    i32.const 0
    select)
  (export "func1" (func 0)))
```

Expected result : 79228162514264337593543950336

Fig. 19. Example of bug found in `[A.7] Unsupported data operatoin`

Summary of answers to RQ3:
(1) We provide 42 test cases in the bug detection framework.
(2) Our crafted bug detection framework can effectively detect bugs in real-world WASM runtimes and provide helpful information to facilitate bug diagnosis and fixing. The bug detection framework uncovered 60 bugs that have never been reported, among which 13 have been confirmed, and 9 have been fixed by runtime developers.
(3) Interestingly, we find that the test cases in our detection framework can trigger more than one type of bug. It further suggests that the summarized bugs show similar patterns among different WASM runtimes.

7 DISCUSSION

7.1 Implications

Given the rapidly increasing popularity of WASM, our study has timely and practical implications for both WASM runtime developers, users and researchers. First, our contribution could help developers dive into and resolve common bugs in WASM runtimes more efficiently. The bug taxonomy in Section 4 could be used as a checklist for the WASM runtime developers to check the commonly occurring bugs. And the fix strategies summarized in Section 5 could be a guideline for developers to fix bugs and optimize the WASM runtimes. For example, it may be hard for a developer to figure out how to solve the *[A.7] Unsupported data operation* symptom, since the bugs could be caused by SIMD instructions, endianness, etc. However, with our guidance, the developer could know how this kind of bugs are usually resolved in practice so that they could find a suitable solution with less trial and error. Furthermore, due to the wide range of WASM runtime bugs that can occur, it is difficult for developers to identify and resolve all of them manually. Therefore, to assist the developers, we construct a bug detection framework that can trigger various bugs we summarized in Section 4 and provide helpful information to facilitate bug fixing.

As an emerging research direction, our study sheds light on future studies on WASM, including automated testing of WASM runtimes, bug fixing with advanced techniques, etc. As for researchers focusing on WASM runtimes, it is promising to detect bugs based on the taxonomy and detector framework we proposed. Moreover, we plan to extend the detector framework by mutating the test cases. Besides, some advanced techniques like fuzzing and differential testing can be adopted as a complement. Besides, our proposed taxonomy is more general by summarizing common issues across different WASM runtimes. In addition to that, we also identified some specific issues in different WASM runtimes that we did not include. These issues are also worth further investigation.

For example, we discovered that WAMR excels in providing thread support among the three WASM runtimes. Developers more frequently utilize WAMR to implement threads compared to wasmer and wasmtime, which in turn exposes more potential thread safety issues in WAMR.

7.2 Threats to Validity

First, our analysis pipeline involves a manual analysis of bugs, which might introduce bias to our observations. To lower the influence of subjective threat, three authors take part in the analysis of bug and fix strategy analysis, discussing the inconsistent issues until reaching an agreement.

Second, our empirical study only targets the most popular WASM runtimes, while there are many WASM runtimes in the wild, and they may pose other kinds of bugs that we did not cover in this paper. Thus, to validate the generalizability of the taxonomy, we select four additional WASM runtimes and classify the closed Github issues with bug labels. As all the bugs in the four WASM runtimes can be classified according to our proposed taxonomy, the threat could be considered neutralized.

Third, it is difficult to ensure that our crafted bug detectors are sound and can cover all the bug patterns of WASM runtimes. To deal with the problem, we evaluate the bug detector's reliability and show that they can trigger bugs in the known WASM runtimes. Nevertheless, for some bug reports, we cannot reproduce them to trigger the bugs the authors mentioned. Since the bug detection framework uses a fixed set of test cases to detect bugs, it is limited once all these runtimes fix the bugs. However, the test cases are constructed based on the issues from wasmer, wasmtime, and WAMR. Interestingly, these test cases also could trigger bugs in wasm3, WasmEdge, wasmer-go, and wasmer-python. In other words, different WASM runtimes show similar bug patterns. The bugs in the aforementioned WASM runtimes could be fixed in the future. But the bug detection framework will always be helpful for the newly developed WASM runtimes to detect new bugs continuously.

8 RELATED WORK

WebAssembly Runtime. WASM runtime has been used in a wide spectrum of applications. Ménétrey et al. proposed WebAssembly trusted runtime, TWINE [67], to execute unmodified, language-independent applications. They leverage Intel SGX to build the runtime environment. Gadepalli et al. [54] proposed a light-weight WASM runtime, Sledge, for the edge computing. Wen et al. propose Wasmachine [75], an OS aiming to efficiently and securely execute WebAssembly applications in IoT and Fog devices with constrained resources. WASM runtime is the fundamental part of various applications. However, there are no studies about bugs in WASM runtimes. We present the first comprehensive study on characterizing and detecting bugs in WASM runtimes.

Other WASM Related Studies. WASM is a promising and newly emerged area. There have been studies on several aspects of WASM, including the WASM execution efficiency [56, 57, 60, 73], WASM compilers [1, 59, 69], WASM binary security [1, 48, 58, 61, 62, 65], etc. As WASM runtime is one of the fundamental components, our study provides timely insights to all stakeholders in the ecosystem.

Empirical Study on bugs. There have been a large number of empirical studies focusing on software bugs across a wide range of applications. For example, Chen et al. [49] studied the faults related to the deployment of DL models on mobile devices; Lu et al. [63] provided the comprehensive real-world concurrency bug characteristic study; Di Franco et al. [52] presented the first comprehensive study of real-world numerical bugs; Wen et al. conducted an empirical study on challenges of application development in serverless computing [76]. Recently, the rapid development of WebAssembly has inspired empirical studies on WebAsssmebly binaries and compilers. For example, Romano et al. [69] conducted an empirical study of bugs in WebAssembly compilers. They investigated 146 bug reports in Emscripten related to the unique challenges WebAssembly compilers encounter compared with traditional compilers. Moreover, Hilbig et al. [58] presented a comprehensive empirical study of 8,461 unique WebAssembly binaries. Following the widely used bug-studying method in the prior studies [49, 76], we applied these bug characterization methods to the bugs in a different domain, i.e., WebAssembly runtimes. Based on the architecture of WASM runtimes, we first constructed four inner categories linked to different components in a WASM runtime (e.g., *Backend Compilation*). Furthermore, we constructed the leaf categories according to the root causes. Zhang et al. [79] conducted an empirical study of TensorFlow program bugs. However, they used bug symptoms, such as error, as the categorization criteria, which could be much easier to understand and follow than our work. With WASM runtime architecture as the categorization criteria, our work could pose a threshold for researchers unfamiliar with WASM runtimes. However, Zhang et al.'s work could not demonstrate the domain-specific characteristics that ours does. Besides, Romano et al. used the stages of compiling high-level language into WASM binaries as the bug categorization criteria in their empirical study [69]. This classification method could better describe the stages when the bug was discovered than our work. Nevertheless, the taxonomy we proposed could better show which part of the system the bug belongs to. Moreover, based on the characterization, we constructed a bug detector for these summarized bugs and found 13 confirmed bugs.

9 CONCLUSION

This paper has presented the first comprehensive study of bugs and the corresponding fix strategies of WASM runtimes. By manually analyzing 311 real-world bugs extracted from the most popular WASM runtimes, we have constructed a taxonomy of bug symptoms with 31 categories, and distilled the fix strategies for them. Based on the knowledge extracted, we further develop a pattern-based bug detection framework to automatically detect bugs across WASM runtimes. By the time of this study, we have identified 60 bugs that have never been reported in the community, and 13 of them have been confirmed by the official developers.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (grant No.2021YFB2701000), the National Natural Science Foundation of China (grants No.62102009 and No.62072046), the Beijing Outstanding Young Scientist Program (grant No.BJJWZYJH01201910001004), and Center for Data Space Technology and System, Peking University. Zhenpeng Chen was supported by the ERC Advanced Grant No.741278 (EPIC: Evolutionary Program Improvement Collaborators).

REFERENCES

[1] 2018. Reverse Engineering WebAssembly. https://www.pnfsoftware.com/reversing-wasm.pdf.
[2] 2019. wasmer issue 830. https://github.com/wasmerio/wasmer/issues/830.
[3] 2020. WAMR issue 1144. https://github.com/bytecodealliance/wasm-micro-runtime/issues/1144.
[4] 2020. wasmer issue 1263. https://github.com/wasmerio/wasmer/issues/1263.
[5] 2020. wasmtime issue 2347. https://github.com/bytecodealliance/wasmtime/issues/2347.
[6] 2021. wasmer-go issue 244. https://github.com/wasmerio/wasmer-go/issues/244.
[7] 2021. wasmer issue 1640. https://github.com/wasmerio/wasmer/issues/1640.
[8] 2021. wasmer issue 2028. https://github.com/wasmerio/wasmer/issues/2028.
[9] 2021. wasmer issue 2187. https://github.com/wasmerio/wasmer/issues/2187.
[10] 2021. wasmer issue 2215. https://github.com/wasmerio/wasmer/issues/2215.
[11] 2021. wasmer issue 2297. https://github.com/wasmerio/wasmer/issues/2297.
[12] 2021. wasmtime issue 2826. https://github.com/bytecodealliance/wasmtime/issues/2826.
[13] 2021. wasmtime issue 2883. https://github.com/bytecodealliance/wasmtime/issues/2883.
[14] 2021. wasmtime issue 3173. https://github.com/bytecodealliance/wasmtime/issues/3173.
[15] 2021. wasmtime issue 3337. https://github.com/bytecodealliance/wasmtime/issues/3337.

[16] 2022. Cranelift Doc. https://hacks.mozilla.org/2020/10/a-new-backend-for-cranelift-part-1-instruction-selection/.

[17] 2022. Cranelift IR Doc. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/ir.md.

[18] 2022. EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine. https://github.com/EOSIO/eos.

[19] 2022. hera - an ewasm (revision 4) virtual machine implemented in C++ conforming to EVMC ABIv9. https://github.com/ewasm/hera.

[20] 2022. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt.

[21] 2022. WAMR issue 1282. https://github.com/bytecodealliance/wasm-micro-runtime/issues/1282.

[22] 2022. WAMR issue 1289. https://github.com/bytecodealliance/wasm-micro-runtime/issues/1289.

[23] 2022. WASI link. https://wasi.dev/.

[24] 2022. Wasm non web usage. https://webassembly.org/docs/non-web/.

[25] 2022. Wasm runtime architecture. https://medium.com/wasm/webassembly-wasm-runtimes-522bcc7478fd.

[26] 2022. wasm3 - The fastest WebAssembly interpreter, and the most universal runtime. https://github.com/wasm3/wasm3.

[27] 2022. wasm3 bug fix commit. https://github.com/wasm3/wasm3/commits/fbbacefeaf28e019244bbfa281fc4dea3dbdedc9.

[28] 2022. wasm3 issue 351. https://github.com/wasm3/wasm3/issues/351.

[29] 2022. wasm3 issue 355. https://github.com/wasm3/wasm3/issues/355.

[30] 2022. WasmEdge bug fix commit. https://github.com/WasmEdge/WasmEdge/commit/4103613ff57341af346f7ff82bd0beb47e798474.

[31] 2022. WasmEdge issue 1711. https://github.com/WasmEdge/WasmEdge/issues/1711.

[32] 2022. WasmEdge Runtime. https://github.com/WasmEdge/WasmEdge.

[33] 2022. wasmer - a fast and secure WebAssembly runtime. https://github.com/wasmerio/wasmer.

[34] 2022. wasmer-go - a complete and mature WebAssembly runtime for Go based on Wasmer. https://github.com/wasmerio/wasmer-go.

[35] 2022. wasmer-python - a complete and mature WebAssembly runtime for Python based on Wasmer. https://github.com/wasmerio/wasmer-python.

[36] 2022. wasmtime - A standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime.

[37] 2022. wasmtime issue 4170. https://github.com/bytecodealliance/wasmtime/issues/4170.

[38] 2022. Wat file. https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm.

[39] 2022. WAVM - a WebAssembly virtual machine, designed for use in non-browser applications. https://github.com/WAVM/WAVM.

[40] 2022. WebAssembly Micro Runtime. https://github.com/bytecodealliance/wasm-micro-runtime.

[41] 2022. WebAssembly system interface Doc. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[42] 2022. WebAssmebly Doc. https://webassembly.org/.

[43] 2023. Emscripten compiler. https://emscripten.org/.

[44] 2023. Supplemental materials. https://github.com/bnmcxlzd/TOSEM2023_Complementary_materials.

[45] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.

[46] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. 2021. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–41.

[47] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. 2018. Automatically classifying posts into question categories on stack overflow. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 211–21110.

[48] Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A Selcuk Uluagac. 2022. A First Look at Code Obfuscation for WebAssembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 140–145.

[49] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 674–685.

[50] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[51] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 509–519.

[52] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE,

509–519.

[53] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142.

[54] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*. 265–279.

[55] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 261–2615.

[56] 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.

[57] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices. *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2* (2018).

[58] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*. 2696–2708.

[59] Eric Holk. 2018. Schism: A Self-Hosting Scheme to WebAssembly Compiler. *Proceedings of the Scheme and Functional* (2018).

[60] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of {WebAssembly} vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 107–120.

[61] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of {WebAssembly}. In *29th USENIX Security Symposium (USENIX Security 20)*. 217–234.

[62] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 410–425.

[63] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 329–339.

[64] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. 2021. WebAssembly modules as lightweight containers for liquid IoT applications. In *International Conference on Web Engineering*. Springer, 328–336.

[65] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. Security chasms of wasm. *NCC Group Whitepaper* (2018).

[66] Pankaj Mendki. 2020. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*. IEEE, 161–166.

[67] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 205–216.

[68] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[69] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An Empirical Study of Bugs in WebAssembly Compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.

[70] Alan Romano and Weihang Wang. 2020. Wasim: Understanding webassembly applications through classification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1321–1325.

[71] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[72] Quentin Stiévenart, David W Binkley, and Coen De Roover. 2022. Static stack-preserving intra-procedural slicing of webassembly binaries. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2031–2042.

[73] Weihang Wang. 2021. Empowering Web Applications with WebAssembly: Are We There Yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1301–1305.

[74] Ziyuan Wang, Dexin Bu, Aiyue Sun, Shanyi Gou, Yong Wang, and Lin Chen. 2022. An Empirical Study on Bugs in Python Interpreters. *IEEE Transactions on Reliability* (2022).

[75] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 1–4.

[76] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 416–428.

[77] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.

[78] Xiuhong Zhang. 2020. *WebAssembly Principles and Core Technologies*. China Machine Press.

[79] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

[80] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884.