A Comprehensive Study of Bugs in Embedded WebAssembly Virtual Machines

Wenlong Zheng Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
zwl21@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—WebAssembly is an emerging platform-independent and safe instruction set architecture, which has been successfully deployed in a wide spectrum of embedded scenarios such as blockchains, edge computing, and Internet-of-Things. WebAssembly virtual machines, like any large software systems, may contain bugs defeating the safety guarantee of WebAssembly. However, few studies have been conducted to investigate bugs in embedded WebAssembly virtual machines. Without such knowledge, virtual machine developers might miss opportunities to further improve virtual machines' qualities, virtual machine testers might fail to detect bugs, and bug detection tool builders might base on wrong assumptions.**

In this paper, we conduct, to the best of our knowledge, the *first* and most *comprehensive* empirical study of bugs in embedded WebAssembly virtual machines. First, we conducted a qualitative study of bugs in four widely used embedded WebAssembly virtual machines: Wasmer, Wasmtime, WAMR, and Wasm3. We identified unique bugs in each of them, investigated their root causes and fixing strategies, then provided a bug reproduction analysis. Second, we conducted a quantitative study on these virtual machines, by a detailed analysis of bug lifecycles, bug testing, and fixing. The empirical results provide interesting findings and insights, such as: 1) we proposed a taxonomy of bug root causes and fixing strategies; and 2) we identified that the bug fixing capability varies greatly in terms of average fixing time, from 16.72 to 98.24 days. We suggest that: 1) virtual machine developers should pay attention to the unique features of WebAssembly, leading to critical bugs; 2) virtual machine testers should improve code coverage to trigger hard-to-detect bugs; 3) bug detection tool builders should develop effective tools to discover potential security defects in WebAssembly virtual machines. We believe our findings and suggestions can help WebAssembly developers, testers, and tool builders by providing better guidelines for WebAssembly studies.

***Index Terms*—Empirical study, WebAssembly virtual machines, Bugs**

I. Introduction

WebAssembly (Wasm) [1] is an emerging platform-independent instruction set architecture and binary distribution format, designed with two important goals of *portability* and *safety*. First, Wasm is designed to be portable, enabling it to be executed by a Wasm virtual machine (VM) on any platform. For example, the Chrome browser has the full support of Wasm via its V8 VM [2]. Specifically, with rapid adoptions of Wasm in embedded scenarios such as gateway API [3], edge computing [4], and Internet-of-Things [5], a wide spectrum of embedded Wasm VMs have been developed.

TABLE I: Embedded Wasm VMs.

Virtual machines	Created	Source	LoC	Release	Stars
Wasmer [6]	2018-10	Rust	1,413,777	70	14.7k
Wasmtime [7]	2016-04	Rust	653,529	59	11.8k
WAMR [8]	2019-04	C	281,503	24	3.6k
Wasm3 [9]	2019-07	C	69,771	7	6k
WasmEdge [4]	2019-06	C++	323,980	63	5.6k
WAVM [10]	2015-08	C	458,083	110	2.4k
Lucet [11]	2019-01	Rust	46,403	7	4.1k
Wazero [12]	2020-05	Go	446,586	11	2.7k
Wasmic [13]	2018-01	Rust	57,085	23	1.1k

Table I presents 9 representative embedded Wasm VMs under active development, with their creation time, implementation languages, code sizes, releases, and popularity.

The second key design goal of Wasm is safety, which indicates that any Wasm program execution might not lead to unintended consequences such as corrupting the hosting environments. To achieve this goal, Wasm incorporates not only a safe type system [14], but also full VM isolations via sandboxing [15]. As a result, it is crucial that Wasm VMs must be implemented correctly and reliably, to guarantee the safe executions of Wasm programs.

Unfortunately, while Wasm VMs should guarantee safety, they, like any large software systems, might contain implementation bugs, defeating their safety guarantees. For example, according to recent common vulnerability exposures (CVEs) [16], Wasmtime [7], a popular Wasm VM implemented in Rust, is reported to contain ten vulnerabilities, which might lead to unexpected consequences such as corrupting memory data by malicious modules [17] [18].

One may speculate that the study of VM reliability is a solved problem, as there have been a significant amount of studies in this direction [19] [20]. However, two issues still remain: first, prior studies on VM security (*e.g.*, JVM [21] or PVM [22]) cannot be applied to the studies of Wasm VMs directly, due to the unique characteristics of Wasm. For example, Wasm introduced a unique feature dubbed *linear memory* [23], whose buggy implementation might lead to memory corruptions such as buffer overflows or heap metadata overwritings [24] [25]. As another example, WASI [26], another unique feature of Wasm, might cause Wasm sandbox escape when implemented incorrectly, leading to security vulnerabilities,

crashes, or incorrect results [27].

Second, while there have been a significant amount of studies on Wasm security [24] [28], few studies have been conducted to investigate the reliability of Wasm VMs. Instead, existing studies just assume the underlying Wasm VMs have been implemented correctly and thus are trustworthy and reliable, whether such an assumption truly holds is still unknown. Furthermore, although recent work [29] has empirically studied Wasm bugs in browser scenarios such as Chrome V8 [30], a comprehensive bug study of *embedded* Wasm VMs is still lacking.

To this end, to study the bugs in embedded Wasm VMs, several key questions remain unanswered: What are novel challenges in developing embedded Wasm VMs? What bugs have been introduced by those challenges? What are the root causes leading to these bugs? How can bugs be reproduced, and what information is needed to reproduce the bugs? How do embedded Wasm VM developers fix these bugs? How long it takes to fix these bugs for different VMs? What are the statistical patterns in the LoC (Line of Code) of test cases that trigger bugs and what are the statistical patterns in the LoC to fix these bugs? Without such knowledge, Wasm VM developers might fail to improve VM implementation quality; VMs testers might fail to trigger bugs effectively; bug detection tool builders might base on wrong assumptions.

Our work. To fill this knowledge gap, this paper presents, to the best of our knowledge, the *first* and most *comprehensive* study of bugs in embedded Wasm VMs by utilizing qualitative and quantitative approaches. This study is performed in three steps. First, to investigate unique bugs in Wasm VMs, we designed and implemented a script to collect bug reports from four widely used embedded Wasm VMs: Wasmer [6], Wasmtime [7], WAMR [8], and Wasm3 [9] automatically, creating a dataset with 1179 bugs.

Second, we performed a qualitative study on these bugs in four VMs to investigate unique challenges in implementing embedded Wasm VMs, such as linear memory management, WASI implementation, and Wasm proposals. We then investigated root causes leading to these challenges, and presented strategies for bug reproductions and fixes.

Third, we conducted a quantitative study on bugs from the qualitative study. During this study, we investigated bug lifecycles, the input code size of bugs as well as the code size of fixing commits.

Empirical results give interesting findings, such as: 1) we identified unique Wasm features bringing challenges to Wasm VM implementations; 2) we revealed root causes leading to Wasm VM bugs; 3) we found that important information in bug reports is often missing, leading to difficulties for bug reproduction; and 4) we found that the bug fixing ability of Wasm Vms varies significantly, with the average fixing time ranging from 16.72 days to 98.24 days, and the average fixing code size ranging from 55.97 LoC to 1157.20 LoC, respectively.

Our findings and empirical results will benefit several audiences. Among others, they: 1) help Wasm VM developers to improve VMs' security and quality; 2) help VM testers to improve testing effectiveness with comprehensive test suites; and 3) help bug detection tool builders to better improve the quality of detection tools.

Contributions. To the best of our knowledge, this work represents the first step toward a comprehensive empirical study of bugs in embedded Wasm VMs. To summarize, our work makes the following contributions:

- **Empirical study.** We presented the first and most comprehensive empirical study on bugs in embedded Wasm VMs.
- **Dataset.** We created and released a dataset of Wasm VMs' bugs with important information such as bug root causes, bug lifecycle, and bug reproduction information, which will benefit future studies in this direction.
- **Findings and insights.** We presented empirical results, findings, suggestions, as well as future challenges and research opportunities.
- **Open source.** We made our dataset, tool, and empirical data publicly available in the interest of open science at https://doi.org/10.5281/zenodo.7866140.

Outline. The outline of this paper is as follows. Section II presents the background for this work. Section III presents the data collection for this study. Sections IV and V present the empirical results qualitatively and quantitatively, respectively. Sections VI and VII discuss implications for this work and threats to validity, respectively. Section VIII discusses the related work, and Section IX concludes.

II. Background and Challenges

To be self-contained, in this section, we present necessary background information on Wasm (§ II-A) and its VM (§ II-B).

A. Wasm

Brief history. Wasm is an emerging universal instruction set architecture, originally developed in 2015 [31], drawing on concepts from NaCl's [32] sandbox execution and asm.js's [33] typing. In 2017, four major browsers: Firefox, Chrome, WebKit, and Edge, agreed on a standard of Wasm and began supporting it [34]. In the following year, the Wasm Core Specification [35] version 1.0 was released, defining the syntax, semantics, and binary and text format. In 2019, the W3C announced that Wasm had become an official Web standard [36] and worked on the WebAssembly System Interface (WASI) [37], a standard that enables Wasm to run outside the browser. In 2022, the Wasm core standard draft version 2.0 [35] was released, introducing new types and directives to improve execution efficiency and expressiveness.

Advanced features. Wasm emphasizes safety, efficiency, and portability. First, to ensure safety, Wasm incorporates secure language features such as strong typing, software fault isolation, secure control flow, and linear memory. Second, Wasm's stack-based abstract instruction set balances space usage and execution efficiency, enabling it to fully utilize underlying hardware capabilities and achieve high execution efficiency close to native code. Third, Wasm is independent of

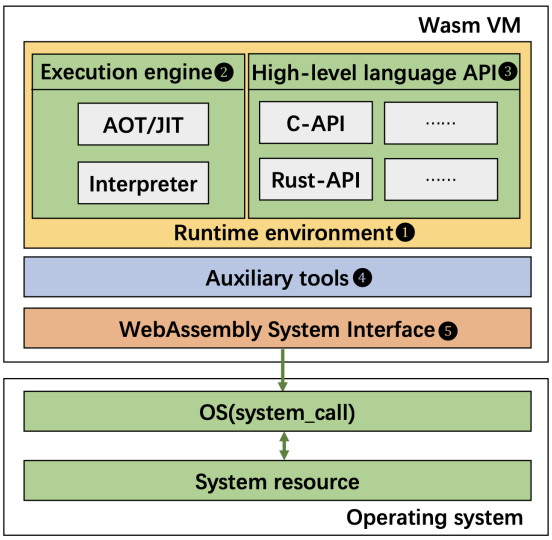


Fig. 1: A representative architecture of a Wasm VM.

high-level languages and specific execution platforms, making it highly portable.

Wide applications. Wasm has wide applications in both Web and non-Web domains. In the Web domain, major browsers have the full support of Wasm, as it is the official W3C standard. In the non-Web domain, with new features such as WASI, Wasm is increasingly used in diverse computing scenarios such as serverless cloud computing [38], IoT [5], embedded devices [39], blockchain [40], edge computing [41], machine learning [42], and game engines [43]. In the future, a desire to secure cloud or edge computing infrastructures without sacrificing efficiency will offer more opportunities for Wasm.

B. Wasm VM

Wasm VM is a runtime environment that executes Wasm programs, consisting of 3 main components: 1) runtime environment; 2) auxiliary tools; and WASI.

To put the discussion of Wasm VM in perspective, Fig. 1 presents a representative architecture of a Wasm VM. First, the runtime environment (❶) consists of two components: the execution engine (❷) and the high-level language API (❸). The execution engine executes the Wasm instructions, either by interpreting them or compiling them to native binary instructions ahead-of-time (AOT) or just-in-time (JIT). The high-level language API serves as the foreign function interface, allowing Wasm VM to interact with programming languages such as C/C++, and Rust.

Second, auxiliary tools (❹) are handy tools provided by Wasm VMs for users' convenience, such as Wasm module cache and Wasm textual file format validation.

Third, the WebAssembly System interface (WASI) (❺) serves as a bridge between the Wasm execution environment and the host environment, providing a standardized interface to access underlying operating system resources, such as file system, networking, and system calls.

III. Data Collection

In this section, we describe the approach for collecting datasets in two main steps: first, selecting the embedded Wasm VMs to be studied (§ III-A); second, identifying and collecting the bugs in these VMs (§ III-B).

A. Selecting Embedded Wasm VMs

We inspected Wasm VM projects on GitHub using the curated awesome-wasm-runtimes list [44], and the top 9 VMs with more than 1,000 stars indicating popularities are given in Table I. It should be noted that our goal in this work is to conduct an empirical study of *embedded* Wasm VMs, hence, the VMs used in browsers and servers such as V8 [30] or SpiderMonkey [45] are not included in this list.

Following prior studies [28], we then selected Wasm VMs according to three criteria: 1) active maintenance; 2) maturity; and 3) popularity. First, we focused on VMs that are still actively maintained. To this end, we pruned out inactive Wasm VMs such as VMIR [46] and Wagon [47], which have not been updated since May 11, 2020. Second, we focused on mature Wasm VMs by filtering out VMs with less than 10,000 lines of code (*e.g.*, warpy [48]) or less than 5 releases (*e.g.*, wazero [12]), following a criterion used in prior work [28]. Third, we focused on popular Wasm VMs by selecting VMs with more than 1,000 stars, following a criterion used in prior work [49]. As a result, we selected 4 Wasm VMs: Wasmer [6], Wasmtime [7], WAMR [8], and Wasm3 [9], which are actively maintained, mature, and popular.

Wasmer. Wasmer [6] is a standalone Wasm VM running outside of the browser, with 14.7k stars and 70 releases. It supports both JIT and AOT and is designed with three internal compilers, providing high performance with a low memory footprint and minimal overhead, making it well suited for use in resource-constrained environments.

Wasmtime. Wasmtime [7] is a standalone VM written in Rust, with 11.8k stars and 59 releases. It has a small footprint and provides an efficient and secure environment for running Wasm modules. It strives to be a highly configurable and embeddable runtime to run on any scale of application and can be implemented on mainstream platforms.

WAMR. WebAssembly Micro Runtime (WAMR) [8] is a lightweight and standalone Wasm VM, with 3.6k stars and 24 releases. It is designed with a small footprint, high performance and highly configurable features, suitable for embedded scenarios such as IoT, edge computing, Trusted Execution Environment (TEE), smart contract, and cloud-native.

Wasm3. Wasm3 [9] is an open-source Wasm VM, with 6k stars and 7 releases, and is designed to be small, fast, and easy to embed in other applications. It supports most of the Wasm specification and can be used on a wide range of embedded platforms.

B. Collecting VM Bugs

We collected bug reports up to December 2022, from the 4 selected Wasm VMs projects' GitHub repositories, using two methods: 1) GitHub Search API [50]; and 2) GitHub REST

TABLE II: Datasets used in our study.

Name	DS1	DS2	DS3	DS4
#Bugs	1179	130	271	387

API [51]. First, we utilized the GitHub Search API to gather closed issues related to Wasm VM. Second, we utilized the GitHub REST API to retrieve all issues and pull requests for the corresponding VM.

We then created, in 4 steps, 4 datasets **DS1** to **DS4**, as presented in Table II. First, to identify all potential bugs, we used keywords such as "bug", "fail", and "defect" to search for all bug-related issues and created a dataset **DS1** with 1179 bugs.

Second, to perform the qualitative study, we identified bugs closely related to unique features of Wasm from **DS1**. To avoid potential bias, we formed an inspection group with 3 graduate students who are familiar with both Wasm and Wasm VMs, to independently conduct a manual inspection of all bugs from **DS1**. Each student read the bug reports independently to determine whether the candidate bug is related to unique Wasm features. Moreover, to ensure the reliability of the inspection results, we adopted the Fleiss' Kappa statistic [52] , which is frequently used to test inter-rater reliability to reach an agreement among all students. As a result, we created a dataset **DS2** with 130 bugs unique to Wasm.

Third, to conduct the quantitative study of bug triggering, we selected bugs that come with test cases from **DS1**, and created a dataset **DS3** with 271 bugs.

Finally, to conduct the quantitative study of bug fixing, we selected bugs having fixing commits from **DS1**, and created a dataset **DS4** with 387 bugs.

We have released these datasets in our open-source corpus: https://doi.org/10.5281/zenodo.7866140.

IV. Study I: Qualitative Study

In this qualitative study, we manually inspected issues that closely relate to Wasm features to investigate the following 4 research questions:

RQ1: Development challenges. What are the unique challenges that developers may encounter while implementing Wasm VMs, and how many bugs do these challenges introduce?

RQ2: Bug root causes. What are the root causes leading to Wasm VMs' bugs?

RQ3: Bug reproducing analysis. What are the challenges that developers may encounter while reproducing bugs and what information is needed to reproduce bugs?

RQ4: Bug fixing strategies. What are the fixing strategies that developers may utilize to fix bugs?

A. RQ1: Development Challenges

Wasm VM developers face a set of challenges that are unique to this new language. We identified these challenges in three steps. First, we proposed a taxonomy to classify these challenges using an inductive coding approach [53] applied to the dataset **DS1** (Table II), based on the underlying root causes. Second, we determined whether the candidate bug is a common VM bug or a bug unique to specific Wasm features, by analyzing bug root causes. Third, we iteratively added and refined categories to form distinct groups of challenges. As a result, we proposed 6 unique Wasm VM development challenges, as presented in Table III.

TABLE III: Challenges for Wasm VM development.

	Development Challenges	#No.	Ratio
1	Memory management	14	10.77%
2	WASI implementation	22	16.92%
3	Wasm proposal	48	36.92%
4	Infrastructure disparity	12	9.23%
5	Architecture discrepancies	21	16.15%
6	Incomplete Wasm VM functionalities	13	10.00%
Total		**130**	**100%**

Challenge 1: Memory management. Wasm VM memory management manages its operation stack and linear memory [23]. We identified 14 bugs (10.77%) related to Wasm VM memory management.

Challenge 2: WASI implementation. The WASI implementations enable VM's interactions with underlying resources such as file systems, POSIX threads, and sockets. We identified 22 bugs (16.92%) related to the WASI implementations in the VMs.

Challenge 3: Wasm proposal. Wasm VMs may not fully support the latest Wasm proposals, due to proposals' frequent updating. We identified 48 bugs (36.92%) related to VM's partial support for Wasm proposals.

Challenge 4: Infrastructure disparity. Wasm VMs are built by leveraging existing tools and infrastructures, and thus any defects of infrastructures can lead to Wasm VM bugs. We identified 12 bugs (9.23%) related to infrastructure disparity.

Challenge 5: Architecture discrepancies. Wasm VM bugs can arise from architecture discrepancies such as different instruction sets or endianness. We identified 21 bugs (16.15%) related to architecture discrepancies.

Challenge 6: Wasm VM functionality incompleteness. Wasm VM bugs can be caused by VM functionality incompleteness, such as missing libraries or incomplete native interfaces. We identified 13 bugs (10.00%) related to Wasm VM functionality incompleteness.

Summary: We proposed a taxonomy to classify Wasm VM development challenges into 6 categories. Among these challenges and bugs, more than one third (36.92%) bugs are due to the VM's incomplete support of Wasm proposals.

B. RQ2: Bug Root Causes

We investigated VMs' bugs to identify and analyze root causes leading to the bugs in 5 steps: 1) we analyzed the conversation on the issue's GitHub page to determine the root cause of the bugs; 2) we grouped similar root causes into generalized challenges, as listed in Table III; 3) we generated root cause categories using a deductive coding approach,

```
//illegal memory access Wasm module
(module
   (func (result i32)
     (i32.load (i32.const 0)))
   (memory 0))
//machine code generated by Warmc
0000000000000000 <aot_func#0>:
  0: 48 8b 47 10  mov 0x10(%rdi),%rax
  4: 48 8b 80 58 01  mov 0x158(%rax),%rax
  b: 8b 00  mov (%rax),%eax
  d: c3  retq
```

Fig. 2: WAMR issue #1371: missing bounds check in code generated by AOT, leading to system crashes.

following existing work [28] [54] [55] [56]; 4) to categorize the bugs, we read the bug reports of the Wasm VM repositories and identified the root cause reported by the developers, and then generalized and extended the existing categories to be more specific to Wasm VMs; and 5) we assigned the bugs to the most direct and relevant category based on the root causes. It should be noted that some root causes may be related to more than one category. For example, if the root cause of a bug is a memory problem that results from the implementation of WASI, we classified it as a memory management bug based on the direct bug symptom.

TABLE IV: Root causes for memory management bugs.

Category	Root cause	#No.
Memory management	Inappropriate memory boundary check	5
	Inappropriate memory allocation	5
	Inappropriate memory resource release	2
	Embedded platform incompatibility	2
Total		**14**

1) **Root Causes for Memory Management Bugs:** We identified 4 root causes leading to memory-management-related bugs, as shown in Table IV. First, 5 bugs are caused by inappropriate linear memory boundary checks, resulting in data corruption. For example, the WAMR issue #1371 as presented in Fig. 2 gives a sample of missing bounds checking in the generated assembly code from the AOT translation, leading to system crashes.

Second, 5 bugs are caused by inappropriate linear memory allocation. For example, as presented in Fig. 3, the Wasm3 VM failed to allocate the maximum memory page size (32768 or 64K bytes) specified by the Wasm standard (issue #353), resulting in linear memory allocation overflow.

Third, 2 bugs are caused by inappropriate memory resource release, resulting in abnormal memory usage and thread problems [57] [58].

Fourth, 2 bugs are caused by memory mode incompatibilities between embedded devices. For example, when allocating large memory on embedded platforms, the Wasm3 VM failed due to its incorrect handling of limited memory on these platforms, leading to system crashes [59].

```
// Wasm3/source/m3_config.h: line 24
define d_m3MaxLinearMemoryPages 32768
```

Fig. 3: Wasm3 issue #353: linear memory allocation failure.

TABLE V: Root causes for WASI implementation bugs.

Category	Root causes	#No.
System interaction	Interaction with file system	12
	Interaction with network	4
	Interaction with clock	1
Standard support	Non-standardized design	2
	Inconsistency with standard	1
Platform discrepancy	Embedded platform discrepancy	1
	Non-embedded platform discrepancy	1
Total		**22**

2) **Root Causes for WASI Implementation Bugs:** We proposed 3 categories of root causes leading to WASI implementation bugs as shown in Table V. First, 17 bugs (12+4+1) are caused by incorrect or incomplete implementation of WASI system interactions [60] [61] [62], such as file systems, networks, and clocks. For example, the Wasmer VM incorrectly places a renamed file outside of the pre-opened directory, breaking the VM's file system sandbox [63].

Second, 3 bugs (2+1) are caused by VM's incomplete support of the WASI standard, leading to VM behaviors confusing both VM developers and VM users [64] [65] [66].

Third, 2 bugs (1+1) are caused by discrepancies of the underlying platforms. For example, the Wasmer VM failed to normalize the line ending on Windows platform, leading to incorrect behaviors on that platform [67].

TABLE VI: Root causes for SIMD proposal bugs.

Category	Root causes	#No.
RA failure[1]	Existing lowering failure	2
Incorrect Results	Incorrect code generation	3
	Missing instruction extension on Vec[2]	1
Program panic/trap	Excessively strong assertion	2
	Vec instruction not implemented	1
	Memory operand missing aligned	1
	Missing bitcast on Vec	1
Total		**11**

[1] "RA" means register allocation.
[2] "Vec" means vector type in SIMD.

3) **Root Causes for Wasm Proposal Bug:** We have identified that Wasm's SIMD (Single-Instruction-Multiple-Data) proposal [68] bugs are the most common among the Wasm proposals bugs. We proposed 3 categories of root causes as shown in Table VI. First, 2 bugs are caused by Wasm VMs' failure to lower SIMD vector values, leading to register allocation errors. For example, the Wasmtime VM failed to lower vector opcode (*e.g.*, the opcode `imul`) correctly, resulting in an error of register usage before being initialized first [69].

Second, 4 bugs (3+1) are caused by incorrect code generation or missing instruction extension on vector types, leading to incorrect execution results. For example, the Wasmtime VM, when running a SIMD-related Wasm program, produced different results due to its failure to migrate vector instructions (*e.g.*, `fabs` and `bnot`) [70].

Third, 5 bugs (2+1+1+1) are caused by excessively strong assertions, unimplemented vector instruction, incorrect operand alignment, or missing bit casting, leading to program panics or traps. For example, as Fig. 4 shows, the Wasmtime VM cannot verify the Wasm program (issue #3099), due to its failure to check the `b8×16.eq` instruction (line 7), leading to runtime errors.

```
1 (module
2   (func (param v128 i32)
3     i8x16.eq
4     v128.store))
```

Fig. 4: Wasmtime issue #3099: Cranelift verifier errors.

TABLE VII: Root causes for infrastructure disparity bugs.

Category	Root cause	#No.
WASI tools	WASI-libc	3
	uvwai	3
	as-wasi	1
Compiling tools	GCC/clang	3
	Lightbeam	1
Library	Rust-crypto	1
Total		**12**

4) **Root Causes for Infrastructure Disparity Bugs:** We proposed 3 categories of root causes leading to infrastructure disparity bugs as presented in Table VII. First, 7 bugs (3+3+1) are caused by VM's WASI-libc support [71], uvwasi [72], and as-wasi [73] [1] [2] [3].

Second, 4 bugs (3+1) are caused by VM's compiler support such as GCC [74] or clang [75], and Lightbeam [76], leading to build failures. For example, the Wasmtime VM cannot be built successfully by the old versions of GCC, due to GCC's incomplete support for the `fp` and `lr` registers [77], leading to an infrastructure disparity bug.

Third, 1 bug is caused by library incompatibility. Specifically, `rust-crypto` [78], a cryptographic library written in the Rust, does not support Wasm VM, leading to linking errors for VM leveraging this library.

5) **Root Causes for Architecture Discrepancy Bugs:** We proposed 2 categories of root causes leading to architecture discrepancy bugs. First, 5 bugs are caused by chip discrepancies. For example, the new M1 chip is incompatible with Intel-based containers, leading to a crash in the Wasmtime VM [79].

Second, 16 bugs are caused by the underlying operating system discrepancies. For example, 2 bugs in the WAMR VM are due to AOT files format discrepancy. Specifically, its AOT mode does not execute normally on macOS because the executable file format on macOS is Mach-O instead of ELF [80].

6) **Root Causes for Incomplete Wasm VM Functionality:** We proposed 3 root causes leading to 13 bugs of incomplete Wasm VM functionalities. First, 9 bugs have been caused by failing to fulfill VM users' customization demands, aiming to enhance functionality and convenience for VM users. For example, a user required WAMR developers to expose the native function type for usage convenience, and WAMR addressed via support for the wasm-c-api [81].

Second, 3 bugs are caused by incomplete support for specific Wasm features such as coroutine scheduling on cache-line [4], multi-module[5] support, and the WASI support for SGX [6].

Third, 1 bug is caused by incomplete support of libc libraries, leading to Wasm program loading errors.

Summary: We proposed a taxonomy to classify the bug root causes into 6 categories. Among others, we revealed: 1) memory boundary checks and allocation bugs account for 71.43% of memory management bugs; 2) the implementation of the WASI file system bugs accounts for 54.55% of WASI implementation bugs; 3) all bugs found in the SIMD proposal were caused by incorrect instruction lowering and selection; 4) WASI tool defects account for 58.33% of other instruction bugs; 5) operating system discrepancy accounts for 76.19% of architecture discrepancy bugs; and 6) bugs failing to fulfill users' demands account for 69.23% in incomplete Wasm VM functionality.

C. RQ3: Bug Reproducing Analysis

Reproducing a bug is a crucial step in the debugging process. However, reproducing certain bugs may require specific inputs, stack traces, and particular versions of the VMs and platforms. We inspected bug reports to determine whether they contain all the critical information or not. Moreover, we analyzed conversations in bug reports to understand the challenges in reproducing bugs.

1) **Information in Bug Reports:** Table VIII presents critical information related to bug reproduction and results for a subset of bug reports due to space limit, and a complete list can be found in our open source at https://doi.org/10.5281/zenodo.7866140. The ☑symbol means that the information is included in the corresponding bug report, whereas the □ symbol indicates that the information is missing.

[1]https://github.com/bytecodealliance/wasmtime/issues/4099
[2]https://github.com/wasm3/wasm3/issues/357
[3]https://github.com/bytecodealliance/wasmtime/issues/2373
[4]https://github.com/bytecodealliance/wasm-micro-runtime/issues/109
[5]https://github.com/bytecodealliance/wasm-micro-runtime/issues/200
[6]https://github.com/bytecodealliance/wasm-micro-runtime/issues/277

These empirical results provide interesting findings and insights. First, important information, such as the relevant Wasm programs (20/32), ground truth (26/32), VM options (21/32), VM versions (21/32), and environments (21/32), are often included or discussed in the bug reports, facilitating the subsequent static bug analysis. Second, some key runtime or language information, such as stack traces (8/32) and high-level language source codes (7/32), are rarely provided, making the subsequent dynamic debugging and bug reproduction difficult.

We identified 2 root causes leading to information missing. First, we speculated that bug submitters might have difficulty in identifying the useful information from massive information in stack traces. Second, bug submitters are prone to provide Wasm binaries rather than high-level source files, leading to high-level source files missing in bug reports.

Overall, our findings emphasize the importance of an automated generation of missing information in bug reports, as bug reporters may not always adhere to the reporting standards.

2) **Bugs that Hard to Reproduce:** If a bug is hard to reproduce, VM developers usually ask bug submitters to provide more information to help them reproduce the candidate bug. Thus, by analyzing the conversations in bug reports, we further investigated what detailed information is often required to reproduce the bug.

TABLE IX presents the empirical results, where the ⊞ symbol indicates that extra information was added after the initial report, as requested by the VM developers.

These empirical results provide interesting findings and insights. First, stack traces (4 cases) and source code (5 cases) are the most common information required to reproduce a bug but are often missing initially, as reflected by Table VIII. Second, the Wasmer VM has the most complete information with respect to bug reports, which has only 1 case to add version information after the initial report.

We identified one key root cause: the Wasmer VM offers a bug report template for submitters, encouraging bug submitters to submit detailed bug information.

Summary: Most bug reports contain incomplete information which may potentially increase the difficulty in reproducing bugs. Among the four VMs analyzed, Wasmer has the most complete bug report information, due to its deployment of a standard bug report template, facilitating bug submitters.

D. RQ4: Bug Fixing Strategies

We investigated the strategies to fix Wasm VMs' bugs, by analyzing the conversations in bug reports to identify any explicit mentions of bug fixes by VM developers. When such information is unavailable, we inspect the bug fixing commits, which is the last commit made before the issue is closed. In the following, we discussed each category of bug fixing strategy separately, and distributed the complete strategy list in our open source.

TABLE VIII: Information included in bug reports.

Wasm VM	ID	Wasm[1]	Stack[2]	GT[3]	Opt.[4]	Ver.[5]	Env.[6]	Src.[7]
WAMR	1501	☑	☐	☑	☑	☑	☑	☐
	1477	☑	☐	☑	☑	☑	☐	☐
	1476	☑	☐	☑	☑	☑	☐	☐
	1299	☑	☐	☑	☑	☐	☑	☐
	1282	☑	☐	☑	☑	☐	☑	☐
	1269	☐	☐	☐	☐	☑	☑	☐
	1230	☑	☐	☐	☐	☑	☐	☐
	1173	☑	☑	☑	☑	☑	☑	☑
Wasmtime	4923	☑	☐	☑	☑	☐	☑	☐
	4875	☑	☑	☑	☐	☑	☑	☐
	4838	☑	☑	☑	☑	☑	☑	☑
	4828	☑	☐	☑	☐	☐	☑	☐
	4705	☐	☐	☑	☑	☑	☑	☑
	4699	☐	☐	☑	☑	☑	☑	☑
	4693	☐	☐	☑	☐	☑	☑	☑
	4677	☑	☐	☑	☑	☑	☑	☐
Wasmr	3614	☐	☐	☑	☑	☑	☐	☑
	3565	☐	☑	☐	☐	☑	☑	☐
	3561	☑	☐	☑	☑	☑	☑	☐
	3510	☐	☑	☐	☑	☑	☑	☐
	3509	☐	☐	☐	☑	☑	☑	☐
	3485	☐	☐	☑	☐	☑	☐	☑
	3481	☐	☑	☐	☐	☐	☐	☐
	3470	☑	☑	☑	☑	☑	☑	☐
Wasm3	321	☑	☐	☑	☑	☐	☑	☐
	313	☑	☑	☑	☑	☑	☑	☐
	295	☐	☐	☑	☐	☐	☐	☐
	258	☑	☐	☑	☑	☑	☑	☐
	227	☑	☐	☑	☑	☐	☐	☐
	218	☑	☐	☑	☑	☐	☐	☐
	198	☑	☐	☑	☐	☐	☐	☐
	124	☐	☐	☑	☐	☐	☐	☐

[1] "Wasm" means a Wasm program is available.
[2] "Stack" means a stack trace is provided.
[3] "GT" means the ground truth of the expected output is listed.
[4] "Opt." means the tool-chain options used are listed.
[5] "Ver." means the version of the runtime used is listed.
[6] "Env." means the information of a running platform is provided.
[7] "Src." means a high-level language source code is provided.

TABLE IX: Bug reports where bugs are difficult to reproduce.

Wasm VM	ID	Wasm	Stack	GT	Opt.	Ver.	Env.	Src.
Wasmtime	4807	☑	☑	☑	⊞	☐	☐	☐
	4234	☐	⊞	☑	☑	☑	☑	⊞
	2552	☑	⊞	☑	☐	☑	☑	☑
	2386	⊞	☑	☑	☐	☐	☐	☐
	1768	☐	⊞	☐	☐	☑	☑	☐
	1323	☐	☐	☑	☑	☑	☐	⊞
WAMR	893	☐	☐	☑	☐	☐	☐	⊞
	518	☐	☐	☑	☐	☐	⊞	⊞
	448	⊞	⊞	☑	⊞	⊞	⊞	⊞
Wasm3	232	⊞	☐	☑	⊞	⊞	☐	☐
	131	☑	☐	☑	☐	⊞	☐	☐
Wasmer	1714	☑	☐	☑	☐	⊞	☐	☐

Refer to Table VIII for the meaning of each table header.
⊞ means the corresponding information was added after the initial report.

1) **Memory Management Bug Fixing:** We proposed 4 categories of bug fixing strategies to fix memory management bugs. First, inappropriate memory boundary check bugs are often fixed by adding or changing memory boundary checks conditions to prevent illegal data access or data overwritten.

For example, the following bug fix in the WAMR VM fixed a buggy range checking.

```
// -- /core/iwasm/aot/aot_runtime.c: line 690
if ((uint8*)module_inst->heap_data.ptr < addr
// ++ /core/iwasm/aot/aot_runtime.c: line 688
if ((uint8*)module_inst->heap_data.ptr <= addr
```

Second, inappropriate memory allocation bugs can be fixed by adjusting memory allocators or allocation strategies. For example, the WAMR VM has modified its allocator to allocate WASI-related environment variables in the global heap, instead of in the Wasm VM local heap, to avoid runtime corruptions.

Third, inappropriate memory release bugs can be fixed by adding necessary release statements. For example, one bug in the Wasm3 VM was fixed by releasing the unused memory [82].

Finally, embedded platform incompatibility bugs can be fixed by dealing with the incompatibilities. For example, the Wasm3 VM leveraged special memory management on IoT devices.

2) **WASI Implementation Bug Fixing:** We proposed 3 categories of bug fixing strategies for WASI implementation bugs. First, we identified 2 strategies for fixing system interaction bugs: 1) granting special permissions for file operations; and 2) implementing the missing WASI features. For example, the Wasmer VM fixed file appending bug by granting the desired append permission [83].

Second, we revealed 2 main categories for fixing Wasm standard violation bugs: 1) for the nuances WASI does not specify, the fixing strategy is to specify clearly implementation-defined behaviors [84]; and 2) for bugs that violate the WASI standard, the strategy is to keep the implementation consistent with the standard [85].

Third, we identified that platform-dependent bugs can be fixed by providing special implementation for the target platform. For example, the Wasm3 VM fixed the function signature bug by updating the function signatures to support ESP32 [86].

3) **Wasm Proposal Bug Fixing:** We proposed 3 categories of strategies to fix Wasm proposal bugs. First, we revealed that bugs can be fixed by supplying missing vector operations. For example, the Wasmtime VM supplied vector operations (*e.g.*, `fabs` and `fnot`), fixing register allocation bugs.

Second, we found that some bugs can be fixed by adjusting the code generation strategy. For example, the Wasmtime VM added bit cast operations to legalize SIMD types (*e.g.*, `b8×16`).

Third, we identified that some bugs can be fixed by removing or relaxing security checks. For example, the Wasmtime VM fixed a runtime panic by removing an incorrect assertion of alignment [87].

4) **Infrastructure Disparity Bug Fixing:** We proposed 3 categories of strategies to fix infrastructure disparity bugs. First, we revealed that bugs can be fixed by incorporating correct library dependencies. For example, the Wasmtime VM fixed a bug caused by a library mismatch, by updating wasi-libc versions.

Second, bugs can be fixed by supporting the dependent infrastructure's version forward or afterward. For example, Wasmtime supported older GCC versions to tackle the build failure.

Third, bugs can be fixed by removing support for unmaintained infrastructures. For example, Wasmtime removed Lightbeam compiler which is unmaintained.

5) **Architecture Discrepancy Bug Fixing:** We proposed 2 categories of bug fixing strategies to fix the architecture discrepancy bugs. First, we found that bugs caused by chip models can often by fixed by adding special support according to the characteristics of the chip. For example, the Wasm3 VM added support for S390X, fixing a big-endian chip related bug [88].

Second, we identified bugs related to operating systems can often be fixed by accounting for the specific operating system features. For example, the Wasm3 VM fixed a function non-existence bug in Ubuntu Xenial 16.04, by supplying an alternative function.

6) **Incomplete Wasm VM Functionality Bug Fixing:** We found 2 bug fixing strategies to fix incomplete Wasm VM functionality bugs. First, we revealed that most bugs can be fixed by adding corresponding functionality support [89].

Second, for those bugs which cannot be fixed in the short term, heuristics of workaround are introduced [90].

Summary: We have proposed 6 bug fixing strategies according to the root causes of the bugs as in the research question **RQ2**, providing actionable guidelines for future bug fixing.

V. Study II: Quantitative Study

In this quantitative study, we inspected bug reports to continue to answer the following 2 research questions:

RQ5: Lifecycle of bugs. How long does it take to fix bugs in corresponding Wasm VMs?

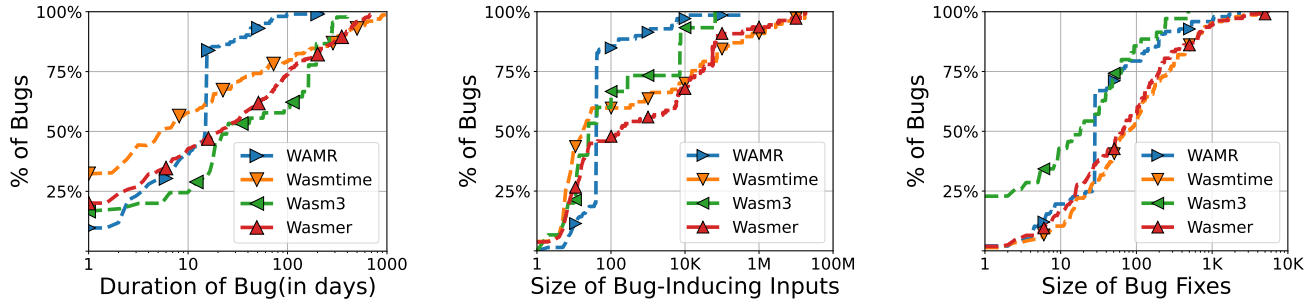
RQ6: Testing and fixing bugs. How many lines of code are needed to trigger and fix bugs?

A. RQ5: Lifecycle of Bugs

To answer **RQ5** by investigating the lifecycles of bugs, we conducted a study on the dataset **DS1** (Table II). We determined the lifecycle of a bug with respect to the time to fix it by analyzing the interval between the bug report's open time and close time. If the bug is reopened, it often indicates that the bug is only partially fixed and still exists, so we used the time of the last closing event as the end of the duration.

Fig. 5a presents the cumulative distribution of bug lifecycles. Within 1 day, Wasmtime, Wasmer, Wasm3 and WAMR successfully fixed 32.75%, 20.12%, 17.78% and 9.71% of their bugs, respectively. Within 10 days, Wasmtime fixed 59.0% of its bugs, while the other three VMs fixed less than 50% of bugs.

These results provide important insights. First, all four VMs fall short of the ideal same-day fix turnaround time [91], leaving considerable space for future improvements.



(a) Cumulative distribution of lifecycles (b) Cumulative distribution of input size (c) Cumulative distribution of fixing size

Fig. 5: The cumulative distribution of lifecycles, input size, and fixing size.

Second, we revealed significant differences in the average lifecycle of bugs in these Wasm VMs. Specifically, bugs in WAMR had an average lifecycle of 16.72 days, while bugs in Wasm3, Wasmtime, and Wasmer had average lifecycles of 93.89, 96.08, and 98.24 days, respectively. We then investigated the root causes and found that the large deviations are attributed to the VM developers' ability to fix intractable bugs or to implement missing Wasm features.

Summary: The bug lifecycles in different Wasm VMs vary significantly. First, all 4 VMs fall short of the ideal fixing time, among which Wasm3 VM only fixed 25.0% bugs within 10 days. Second, the average bug lifecycles vary significantly, from 98.24 days to 16.72 days. These results demonstrate considerable improvement space for these VMs.

B. RQ6: Triggering and Fixing Bugs

To answer **RQ6** by investigating the code sizes of triggering and fixing bugs, we conducted our study based on the **DS3** and **DS4** datasets (Table II), with 271 and 387 bugs, respectively.

1) **Size of Bug-Inducing Test Inputs:** Fig. 5b shows the distribution of code sizes of bug-inducing Wasm inputs in the bug reports.

The empirical results give interesting findings and insights. First, 73 (26.94%) bug-inducing inputs in all four VMs are less than 10 lines of code, and 166 (61.25%) inputs are less than 100 lines of code, indicating most bug-inducing inputs are of manageable sizes.

Second, we revealed that, during the bug fixing process, bug-inducing code sizes are often trimmed to facilitate bug localization. For example, for a Wasmtime bug [7], the Wasm program initially provided is large (454 LoC). Then, multiple posts on the same issue gradually minimize the size of the bug inputs (7 LoC).

Our observations shed light on the effects of bug sizes on bug localization or debugging.

2) **Size of Bug Fixing:** Fig. 5c presents the distributions of sizes for the bug fixings.

The empirical results give interesting findings and insights. First, for all 4 VMs, 17.3% of all bugs can be fixed within 10 lines of code (Wasm3: 42.85%, WAMR: 19.58%, Wasmtime: 10.34%, and Wasmer: 16.36%), and 63.73% of all bugs can be fixed in less than 100 lines of code (Wasm3: 85.71%, WAMR: 79.4%, Wasmer: 56.36%, and Wasmtime: 53.10%, respectively).

Second, Wasm3 takes 55.97 lines of code on average to fix bugs, whereas the other three VMs take more than 100 lines of code (Wasmtime: 273.76, Wasmer: 1,157.20, and WAMR: 124.10, respectively).

We then investigated the root causes for such deviations and found one key reason: some VM developers tend to incorporate many small changes into a single commit, leading to large commit sizes. For example, in the Wasmer VM, we have identified a large commit (with 12,591 LoC additions and 12,682 LoC deletions) containing not only code to implement new Context API but also the missing test suites [92].

Summary: Bug-inducing codes are often of manageable sizes (61.25% of which are less than 100 lines of code). Except for Wasm3 (55.97 LoC), the bug fixing code for the other 3 VMs are large (from 124.10 LoC to 1,157.20 LoC).

VI. Implications

This paper presents the first and most comprehensive empirical study on bugs in embedded Wasm VMs. In this section, we discuss some implications of this work, along with some important directions for future research.

For Wasm VM developers. Results in this work provide Wasm VM developers with important insights into improving the quality and security of Wasm VMs. On the one hand, the categories of bugs we proposed provide guidelines for developers to avoid common pitfalls. On the other hand, the bug root causes we investigated provide actionable references for effective bug fixing.

For Wasm VM testers. Results in this work provide Wasm VM testers with valuable bug causes and statistical analysis to conduct tests more effectively. On the one hand, the root causes analyzed in the qualitative study can serve as a reference to testers to impose a more comprehensive test on susceptible modules in Wasm VM. On the other hand, the statistical analysis in the quantitative study provides guidelines

[7]https://github.com/bytecodealliance/wasmtime/issues/3160

for testers to submit comprehensive reports, mitigating the difficulty of bug reproduction and further bug fixing.

For bug detection tools builders. Results in this work assist Wasm tool developers in enhancing their existing tools and creating new ones that address the challenges identified in this study. On the one hand, the bug datasets we collected can serve as a ground truth benchmark to evaluate tools' effectiveness. On the other hand, the root causes we investigated facilitate development of tools for quicker identification and resolution of Wasm-related bugs in Wasm VMs.

VII. Threats to Validity

As in any empirical study, there are threats to the validity of our work. We attempt to remove these threats where possible and mitigate the effect when removal is not possible.

Wasm VMs. In this paper, we have selected 4 Wasm VMs according to their sizes, forks, popularity (stars), and issues, following selection criteria in prior work [49]. On top of that, we have good reason to believe the Wasm VMs we select are representative based on the solid criteria in prior work. Although some VMs listed in Table I resume maintenance and popularity, such as WasmEdge [4], we will further apply the same methodologies to them in future studies.

Datasets. In this paper, we constructed different datasets for different research questions to mitigate data disturbance. In the qualitative study, we focused on unique Wasm VM development challenges, hence we ruled out those issues that were found to be irrelevant to Wasm after manual inspection. In the quantitative study, we focused on bug fixing ability among Wasm VMs, so we selected bugs that include test cases (**DS3**) and fixing commits (**DS4**), and pruned out false positive bug reports. On top of that, we believe our datasets are representative and trustworthy.

Analysis methodology. In this paper, we manually analyzed the root causes of bugs through an iterative and comprehensive analysis and categorized them by utilizing a widely used inductive method [53] to mitigate subjective bias. However, it is impossible to remove the bias subject to a person's preference. To further mitigate it, we formed a group of three experienced investigators to analyze these bugs separately and discussed inconsistent results until an agreement was reached based on Fleiss' Kappa statistic [52]. On top of that, we believe our study results are convincing and reliable.

VIII. Related Work

There is a significant amount of research effort on Wasm security. However, the work in this paper stands for a novel contribution to this field.

Wasm binary security. There have been many studies on Wasm binary security. Lehmann et al. [93] [24] conducted studies on Wasm security. To enhance Wasm security, Arteaga et al. [94] proposed CROW, a system that uses code diversification to mitigate malicious attacks. Narayan et al. [95] proposed Swivel, a static security enhancement framework based on compiler technology, to defend against spectre attacks [96]. However, the focus of prior work is mainly on the analysis of Wasm binaries security enhancement, instead of on the security of the Wasm VMs.

Wasm runtime security enhancement. There are existing studies on Wasm runtime security. To enhance code protection, Sun et al. [97] proposed the runtime protection framework SELWasm. Menetrey et al. [98] proposed Twine, which provides a trusted execution environment for Wasm VMs by leveraging the trusted execution capabilities such as Intel SGX [99] [100]. However, prior work focused on the analysis vulnerabilities, but we conducted a comprehensive bug study in embedded Wasm VMs.

Wasm vulnerability analysis. There have been many studies on Wasm analysis tools. To detect vulnerabilities in Wasm, static analysis, dynamic analysis, and a combination of them, are utilized. For static analysis, Stiévenart et al. [101] proposed an information flow analysis algorithm for Wasm programs, and Lopes et al. [102] proposed Wasmati, a detection framework based on the code property graph. For dynamic analysis, Chen et al. [103] proposed a fuzzing framework called Wasai for Wasm smart contracts. Szanto et al. [104], and Fu et al. [105] performed taint analysis to track the propagation of data and detect possible input vulnerabilities. For mixture analysis, Sun et al. [106] proposed WASP, a deep learning-based detection framework based on Wasabi [107]. Furthermore, many studies have been conducted to enhance Wasm security [108] [109] [110]. However, the focus of these tools is mainly on the detection of Wasm binaries vulnerabilities, instead of the vulnerabilities in Wasm VMs.

Empirical studies. There have been many empirical studies on software security. For example, Tan et al. [111] conducted an empirical study on open-source projects to identify bug root causes and impact. Eyolfson et al. [112] conducted an empirical study on open-source projects to analyze correlations between bugginess and commit time. Wang et al. [29] conducted an empirical study on Wasm runtimes to analyze bug root causes. Romano et al. [28] conducted an empirical study on Wasm compilers to analyze bug root causes and fixing strategies. However, prior work focused less on a special application scenario, but we conducted a comprehensive empirical study on Wasm-related bugs in Wasm VMs and their impact on embedded devices.

IX. Conclusion

In this work, we present the first empirical study of bugs embedded Wasm VMs. By utilizing both qualitative and quantitative approaches, we studied 4 widely used Wasm VMs. In the qualitative study, we investigated development challenges, bug root causes, bug reproducing and fixing strategies, and proposed a taxonomy of root causes in 6 categories. We further provided constructive suggestions to guide Wasm VMs bug fixing. In the quantitative study, we analyzed the bug lifecycles, code sizes for bug triggering and fixing, among different Wasm VMs. We further pointed out potential research opportunities and future research directions based on the research findings.

REFERENCES

[1] "Webassembly," https://webassembly.org/.

[2] "Webassembly dynamic tiering ready to try in chrome 96 · v8," https://v8.dev/blog/wasm-dynamic-tiering.

[3] "Apache/apisix: The cloud-native api gateway," https://github.com/apache/apisix.

[4] "Wasmedge/wasmedge: Wasmedge is a lightweight, high-performance, and extensible webassembly runtime for cloud native, edge, and decentralized applications. it powers serverless apps, embedded functions, microservices, smart contracts, and iot devices." https://github.com/WasmEdge/WasmEdge.

[5] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *Web Engineering*, ser. Lecture Notes in Computer Science, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds. Cham: Springer International Publishing, 2021, pp. 328–336.

[6] "Wasmerio/wasmer:the leading webassembly runtime supporting wasi and emscripten," https://github.com/wasmerio/wasmer.

[7] "Bytecodealliance/wasmtime: A fast and secure runtime for webassembly," https://github.com/bytecodealliance/wasmtime.

[8] "Bytecodealliance/wasm-micro-runtime: Webassembly micro runtime (wamr)," https://github.com/bytecodealliance/wasm-micro-runtime.

[9] "Wasm3/wasm3:the fastest webassembly interpreter, and the most universal runtime," https://github.com/wasm3/wasm3.

[10] "Wavm/wavm: Webassembly virtual machine," https://github.com/WAVM/WAVM.

[11] "Bytecodealliance/lucet: Lucet, the sandboxing webassembly compiler." https://github.com/bytecodealliance/lucet.

[12] "Wazero: The zero dependency webassembly runtime for go developers," Tetrate Labs, Nov. 2022.

[13] "Paritytech/wasmi: Webassembly (wasm) interpreter." https://github.com/paritytech/wasmi.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, Jun. 2017, pp. 185–200.

[15] "Security - webassembly," https://webassembly.org/docs/security/.

[16] "Cve-cve," https://cve.mitre.org/.

[17] "Cve-cve-2023-26489," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26489.

[18] "Cve-cve-2022-39392," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-39392.

[19] J. S. Reuben, "A survey on virtual machine security," p. 5.

[20] J.-W. Maessen, V. Sarkar, and D. Grove, "Program analysis for safety guarantees in a java virtual machine written in java," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering - PASTE '01*. Snowbird, Utah, United States: ACM Press, 2001, pp. 62–65.

[21] P. H. Hartel, "Formalising the safety of java, the java virtual machine and java card," p. 52.

[22] C. Jiang, B. Hua, W. Ouyang, Q. Fan, and Z. Pan, "Pyguard: Finding and understanding vulnerabilities in python virtual machines," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2021, pp. 468–475.

[23] "Design/semantics.md at main · webassembly/design."

[24] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security*, 2020, p. 19.

[25] J. Bergbom, "Memory safety: Old vulnerabilities become new with webassembly," Tech. Rep., 2018.

[26] "Webassembly/wasi: Webassembly system interface," https://github.com/WebAssembly/WASI.

[27] "Renaming a file unexpectedly places it outside of pre-opened directory · issue #1759 · wasmerio/wasmer."

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

[29] Y. Wang, "A comprehensive study of webassembly runtime bugs."

[30] "V8 javascript engine," https://v8.dev/.

[31] "Going public launch bug · issue #150 · webassembly/design," https://github.com/WebAssembly/design/issues/150.

[32] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code."

[33] "Asm.js," http://asmjs.org/.

[34] "Roadmap - webassembly," https://webassembly.org/roadmap/.

[35] "Webassembly core specification," https://www.w3.org/TR/wasm-core-1/.

[36] "World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation," https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.

[37] "Standardizing wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.

[38] "Webassembly on cloudflare workers," http://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Oct. 2018.

[39] R. Gurdeep Singh and C. Scholliers, "Warduino: A dynamic webassembly virtual machine for programming microcontrollers," *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pp. 27–36, Oct. 2019.

[40] "Diving into ethereum's virtual machine(evm): The future of ewasm — hackernoon," https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy.

[41] "Serverless edge compute solutions — fastly," https://www.fastly.com/products/edge-compute.

[42] "Wasi-nn," WebAssembly, Apr. 2023.

[43] "Viry3d," http://www.viry3d.com/.

[44] "Appcypher/awesome-wasm-runtimes: A list of webassemby runtimes," https://github.com/appcypher/awesome-wasm-runtimes.

[45] "Spidermonkey — firefox source docs documentation," https://firefox-source-docs.mozilla.org/js/index.html.

[46] "Andoma/vmir: Virtual machine for intermediate representation," https://github.com/andoma/vmir.

[47] "Go-interpreter/wagon: Wagon, a webassembly-based go interpreter, for go." https://github.com/go-interpreter/wagon.

[48] "Kanaka/warpy: Webassembly interpreter in rpython," https://github.com/kanaka/warpy.

[49] W. Li, M. Jiang, X. Luo, and H. Cai, "Polycruise: A cross-language dynamic information flow analysis."

[50] "Search," https://ghdocs-prod.azurewebsites.net/en/rest/search.

[51] "Github rest api," https://ghdocs-prod.azurewebsites.net/en/rest.

[52] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. 76, pp. 378–382, 1971.

[53] "Synthesizing qualitative research in software engineering — proceedings of the 40th international conference on software engineering," https://dl.acm.org/doi/10.1145/3180155.3180235.

[54] "Have things changed now? — proceedings of the 1st workshop on architectural and system support for improving software dependability," https://dl.acm.org/doi/abs/10.1145/1181309.1181314.

[55] "Bug characteristics in open source software — springerlink," https://link.springer.com/article/10.1007/s10664-013-9258-8.

[56] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 101–110.

[57] "Memory leak in runtime or environment · issue #203 · wasm3/wasm3," https://github.com/wasm3/wasm3/issues/203.

[58] "On android platform,wasm-micro-runtime initialized but not destroied,multiple threads creates will be failed for the second time. · issue #513 · bytecodealliance/wasm-micro-runtime," https://github.com/bytecodealliance/wasm-micro-runtime/issues/513.

[59] "Function "evaluateexpression" cause stack overflow in iot os · issue #186 · wasm3/wasm3," https://github.com/wasm3/wasm3/issues/186.

[60] "Webassembly/wasi-filesystem: Filesystem api for wasi," https://github.com/WebAssembly/wasi-filesystem.

[61] "Webassembly/wasi-clocks: Clocks api for wasi," https://github.com/WebAssembly/wasi-clocks.

[62] "Webassembly/wasi-poll," https://github.com/WebAssembly/wasi-poll.

[63] "Renaming a file unexpectedly places it outside of pre-opened directory · issue #1759 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/issues/1759.

[64] "Wasmer_wasi: Unable to set env values containing = (equal sign) · issue #1708 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/issues/1708.

[65] "Operations fail when the root (/) is mapped to a directory · issue #2098 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/issues/2098.

[66] "Fd_datasync on stdout returns 0 · issue #1703 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/issues/1703.

[67] "Line endings in wasi on windows · issue #1665 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/issues/1665.

[68] "Webassembly/simd: Branch of the spec repo scoped to discussion of simd in webassembly," https://github.com/webassembly/simd.

[69] "Register allocation failure on x86_64 with simd enabled · issue #3160 · bytecodealliance/wasmtime," https://github.com/bytecodealliance/wasmtime/issues/3160.

[70] "X64: Different results with simd depending on optimization level · issue #3336 · bytecodealliance/wasmtime," https://github.com/bytecodealliance/wasmtime/issues/3336.

[71] "Wasi libc," WebAssembly, Apr. 2023.

[72] "Uvwasi," Node.js, Apr. 2023.

[73] F. Denis, "As-wasi," Apr. 2023.

[74] "Gcc-mirror/gcc," https://github.com/gcc-mirror/gcc.

[75] "The llvm compiler infrastructure," LLVM, Apr. 2023.

[76] "Mozilla/lightbeam: Orignal unmaintained version of the lightbeam extension. see lightbeam-we for the new one which works in modern versions of firefox." https://github.com/mozilla/lightbeam.

[77] "175512 – [arm64] use x29 and x30 instead of fp and lr to make gcc happy," https://bugs.webkit.org/show_bug.cgi?id=175512.

[78] Palmer, "Rust-crypto," Apr. 2023.

[79] "Mac m1 docker crash · issue #3203 · bytecodealliance/wasmtime," https://github.com/bytecodealliance/wasmtime/issues/3203.

[80] "Wamrc can not output aot file normally on macos · issue #267 · bytecodealliance/wasm-micro-runtime," https://github.com/bytecodealliance/wasm-micro-runtime/issues/267.

[81] "Expose type information of wasm functions · issue #286 · bytecodealliance/wasm-micro-runtime," https://github.com/bytecodealliance/wasm-micro-runtime/issues/286.

[82] "Remove memory leak caused by dangling retfunctypes in m3environment by axic · pull request #204 · wasm3/wasm3," https://github.com/wasm3/wasm3/pull/204.

[83] "Fix wasi append bug, add test by markmccaskey · pull request #939 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/pull/939.

[84] "Fix(wasi) allow the '=' sign in the environment variable value. by hywan · pull request #1762 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/pull/1762.

[85] "Audit constants in the documentation · issue #132 · webassembly/wasi," https://github.com/WebAssembly/WASI/issues/132.

[86] "Update link functions in m3_api_esp_wasi.c by robinvanemden · pull request #142 · wasm3/wasm3," https://github.com/wasm3/wasm3/pull/142.

[87] "Remove unnecessary, too strict assertion. fix for 3161. by jlb6740 · pull request #3209 · bytecodealliance/wasmtime," https://github.com/bytecodealliance/wasmtime/pull/3209.

[88] "Wasm3 mis-parses i32 in s390x (big-endian) · issue #321 · wasm3/wasm3," https://github.com/wasm3/wasm3/issues/321.

[89] "Enable post-mvp feature wasm-c-api by lum1n0us · pull request #315 · bytecodealliance/wasm-micro-runtime," https://github.com/bytecodealliance/wasm-micro-runtime/pull/315.

[90] "Removing fprintf · issue #121 · wasm3/wasm3," https://github.com/wasm3/wasm3/issues/121.

[91] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrücken Germany: ACM, Jul. 2016, pp. 294–305.

[92] "Implement new context api for wasmer 3.0 by amanieu · pull request #2892 · wasmerio/wasmer," https://github.com/wasmerio/wasmer/pull/2892.

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

[94] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *Proceedings 2021 Workshop on Measurements, Attacks, and Defenses for the Web*. Virtual: Internet Society, 2021.

[95] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening webassembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[96] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 1–19.

[97] J. Sun, D. Cao, X. Liu, Z. Zhao, W. Wang, X. Gong, and J. Zhang, "Selwasm: A code protection mechanism for webassembly," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. Xiamen, China: IEEE, Dec. 2019, pp. 1099–1106.

[98] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for webassembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. Chania, Greece: IEEE, Apr. 2021, pp. 205–216.

[99] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64.

[100] V. Costan and S. Devadas, "Intel sgx explained."

[101] Q. Stievenart and C. D. Roover, "Compositional information flow analysis for webassembly programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 13–24.

[102] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, Jul. 2022.

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

[104] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," Jul. 2018.

[105] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," Feb. 2018.

[106] P. Sun, L. Garcia, Y. Han, S. Zonouz, and Y. Zhao, "Poster: Known vulnerability detection for webassembly binaries," Tech. Rep., Apr. 2021.

[107] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 1045–1058.

[108] "Wasmview — proceedings of the acm/ieee 42nd international conference on software engineering: Companion proceedings," https://dl.acm.org/doi/10.1145/3377812.3382155.

[109] A. Romano and W. Wang, "Wasim: Understanding webassembly applications through classification," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2020, pp. 1321–1325.

[110] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Jan. 2019.

[111] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools."

[112] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1009–1039, Aug. 2014.