2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00041

A Comprehensive Study of WebAssembly Runtime Bugs

Yue Wang[a], Zhide Zhou[a], Zhilei Ren[a, b*], Dong Liu[a], He Jiang[a, c]
[a] School of Software, Dalian University of Technology, Dalian, China
[b] Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[c] Key Laboratory for Artificial Intelligence of Dalian, Dalian, China
wang_yue11@163.com, cszide@gmail.com, dongliu@mail.dlut.edu.cn, {zren, jianghe}@dlut.edu.cn

***Abstract*—WebAssembly runtime is the infrastructure for executing WebAssembly, which is widely used as an execution engine by web browsers or blockchain platforms. Bugs in the WebAssembly runtime can lead to unexpected behavior and even security vulnerabilities in any application that relies on it. Therefore, to aid developers in understanding the WebAssembly runtime, a thorough investigation of bugs in the WebAssembly runtime should be conducted. To accomplish this, we carry out the first empirical analysis of 867 real bugs across four popular WebAssembly runtimes (V8, SpiderMonkey, Wasmer, and Wasmtime). We analyze the WebAssembly runtime bug characteristics based on their root causes, symptoms, bug-fixing time, and the number of files and lines of code involved in the bug fixes. Here are a few major research findings: 1) Incorrect Algorithm Implementation accounts for 25.49% of WebAssembly runtime bugs, the most prevalent of all root causes; 2) The most prevalent symptom is Crash, which accounts for 56.86% of WebAssembly runtime bugs; 3) At the median, the bug-fixing time are 13, 4, 5, and 6 days for V8, SpiderMonkey, Wasmer, and Wasmtime respectively; 4) Over 50% of bug fixes in the four WebAssembly runtimes involve only one file, while more than 90% of bug fixes involve no more than 8 files; 5) The median source code lines for bug fixes for V8, SpiderMonkey, Wasmer, and Wasmtime are 18.5, 14, 26, and 36 lines, respectively. Overall, our research summarizes 18 findings and discusses the broad implications for WebAssembly runtime bug detection, localization, debugging, and repair based on the key findings.**

***Index Terms*—WebAssembly Runtime, Empirical Study, WebAssembly, Bug Characteristics.**

I. Introduction

WebAssembly is a portable and executable bytecode format, which provides compilation targets for advanced languages such as C++, C#, and Rust. Since its release in 2017, WebAssembly has been widely used in various scenarios, such as cryptocurrency [1], edge computing [2], and the internet of things [3]. WebAssembly runtime is the infrastructure for executing WebAssembly, which is widely used as an execution engine by web browsers or blockchain platforms. For example, the V8 engine inside Chrome [4] is an actual execution environment for WebAssembly bytecode. Specifically, WebAssembly bytecode is parsed and converted by the WebAssembly runtime into machine code relevant to the host platform for execution.

The development and use of the WebAssembly runtime have gradually become a popular trend [2], [4]–[11]. WebAssembly runtime has a decisive impact on the accuracy of all WebAssembly programs running on the WebAssembly runtime. Therefore, ensuring the correctness and robustness of the WebAssembly runtime implementation is becoming increasingly important. Like other applications, the WebAssembly runtime is subject to bugs in its work. Based on the basic role of the WebAssembly runtime, bugs in the WebAssembly runtime can lead to unexpected behavior and even security vulnerabilities in applications that rely on it [12]–[14]. In practice, however, not all application developers can detect WebAssembly runtime bugs on time. Especially inexperienced developers may first assume that the behavior caused by the defect is caused by their own programming. Therefore, to better understand, detect, and fix WebAssembly runtime bugs, the characteristics of WebAssembly runtime bugs need to be analyzed.

This prompts us to conduct the first empirical study of WebAssembly runtime bug characteristics to advance the understanding of WebAssembly runtime bugs. We selected WebAssembly runtimes V8 [4], and SpiderMonkey [15], which are in two mainstream browsers, and two popular standalone runtimes Wasmer [16], and Wasmtime [17] as research objects. In total, we studied 867 real bugs in the four WebAssembly runtimes. For each bug, we examined its issue messages, comments, commit messages, or linked pull request messages. The purpose of our research is to find the answers to the following research questions.

RQ1: How are the root causes of WebAssembly runtime bugs distributed? Root causes help researchers gain insight into the nature of bugs. In this question, we first categorize root causes for bugs based on a systematic process and 16 root causes are identified (described in Section III). We then analyze the root cause distribution of these bugs.

RQ2: How are the symptoms of WebAssembly runtime bugs distributed? Symptoms aid in understanding the consequences of bugs and aid in designing test methods with different test oracles. The symptoms are classified in a similar process as the root cause, with six categories of symptoms identified (described in Section III). We then further analyze the distribution of symptoms for these bugs.

RQ3: What is the connection between WebAssembly runtime bug root causes and symptoms? On the basis of RQ1 and RQ2, we further dig into the obvious mapping

*Corresponding author

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00041

connection between some root causes and special symptoms, which can deepen a more comprehensive understanding of bugs for developers.

RQ4: How long do WebAssembly runtime bugs take to fix? We not only examine the distribution of bug fixes over time for WebAssembly runtimes but also further compute statistics on bug-fixing times for different WebAssembly runtimes, such as mean bug-fixing time and bug-fixing time.

RQ5: How many files and lines of code must be changed to fix a WebAssembly runtime bug? We investigate the fix of WebAssembly runtime bugs, i.e. the files and code lines involved in the bug fixes. Not only did we analyze the distribution of bug fixes across different WebAssembly runtimes in files and lines of code, we further analyze the correspondence between the root cause or symptom and the number of files or lines of code involved in the bug fix.

RQ6: Do the bugs of different WebAssembly runtimes have anything in common? The study of the commonalities between different WebAssembly runtime bugs can help developers design more general testing and debugging techniques.

Based on our empirical results, 18 important findings are revealed, and the specific information about these findings is described in the relevant section of this paper (Section IV). In addition, we discuss their broad implications for WebAssembly runtime bug detection, localization, debugging, and repair in light of some important findings (Section V). For example, Findings 1 and 8 show that *Incorrect Algorithm Implementation* is the most frequent root cause and can result in a variety of symptoms, suggesting that developing techniques that can automatically detect and locate these bugs is a promising research direction.

Our research can aid in the better understanding of WebAssembly runtime bugs by researchers and developers. To sum up, our paper contributes the following:

- We present the first empirical study of WebAssembly runtime bugs, based on 867 real bugs in four widely-used and distinct WebAssembly runtimes.
- We categorize the root causes and symptoms of WebAssembly runtime bugs and further analyze WebAssembly runtime bug fix information (bug-fixing times, files, and code lines).
- We summarize 18 findings and discuss the broad implications for WebAssembly runtime bug detection, localization, debugging, and repair based on some key findings.
- We have released the dataset and code publicly available online[1] for others to copy or reproduce, or even do further research based on our work.

II. WebAssembly Runtimes

The WebAssembly runtime is a program responsible for translating WebAssembly binary instructions into native CPU machine code. Generally speaking, there are three ways to perform this translation: interpret execution, ahead-of-time (AOT) compilation to the native executable, and just-in-time (JIT) compilation to native machine code at runtime. As shown in Fig. 1, WebAssembly runtime can choose to implement an interpreter or an AOT/JIT compiler to accomplish one or more translation methods. The WebAssembly runtime can provide self-developed optimizing compilers, but usually, developers reuse existing backend compilers such as LLVM [18]. In addition, some WebAssembly runtimes offer several translation methods to balance compilation time and code quality. For example, when browsing the web, using JIT compilation is a reasonable choice because a fast startup is important to provide a good user experience.

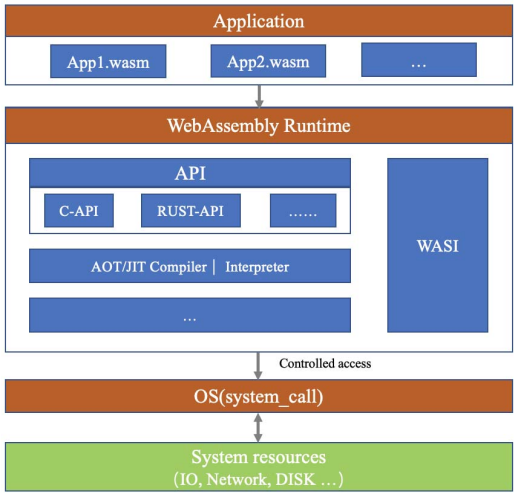


Fig. 1. The architecture of WebAssembly runtimes.

The existing WebAssembly runtime runs as a user space program that sits between the WebAssembly application and the underlying operating system [7], as shown in Fig. 1. In this architecture, WebAssembly applications that want to access IO or external resources need to call the WebAssembly System Interface (WASI) [19], which defines a group of POSIX-like interfaces to operating systems. In addition, the WebAssembly runtime can be embedded in different host programs to achieve the goal of using WebAssembly by the host program. Different host programs can parse and execute WebAssembly programs through the public APIs provided by the WebAssembly runtime.

In this study, four popular WebAssembly runtimes were selected for investigation, namely, V8 [4], SpiderMonkey [15], Wasmer [16], and Wasmtime [17]. Although they are all constructed using the aforementioned architecture, the four WebAssembly runtimes are each distinct. For example, implementations using different programming languages, embedded implementations for Web environments or standalone implementations for non-Web environments, and different development organizations.

[1] https://github.com/Wang11Yue/WebAssembly_Runtime_Bugs

III. METHODOLOGY

A. Selection of WebAssembly Runtimes

As WebAssembly is adopted by more and more domains, the ecosystem of WebAssembly runtime is also expanding. In addition to the WebAssembly runtimes V8 (Chrome), SpiderMonkey (Firefox), JavaScriptCore (WebKit), and Chakra (Edge), which are born in the four major browsers [5]. A large number of standalone WebAssembly runtimes [6] that support WebAssembly parsing and execution are also scrambling to emerge.

We focus on the diversity and popularity of the project, and the number of bugs in the project in this study. Therefore, we select WebAssembly runtimes V8 [4], SpiderMonkey [15], Wasmer [16], and Wasmtime [17] as research objects.

B. Collection of Bugs and Bug-Fixing Commits

1) Collection of Bugs: To investigate the characteristics of WebAssembly runtime bugs, we follow existing work [20]–[23] and first collect the fixed bugs from the WebAssembly runtime issue tracking system, i.e., extracting issues with status closed and fixed. However, the specific implementations of the issue tracking system of V8, SpiderMonkey, Wasmer, and Wasmtime are different, so different rule filters are applied to them to obtain an initial list of WebAssembly runtime bugs called $BugSet1$ (third column of Table I). This process is carried out as follows:

Collecting V8 Bugs. We first search for WebAssembly runtime bugs that satisfy the following screening criteria in the issue tracking system for the V8 engine[2]: (1) the type of issue is "Bug", (2) its status is "Fixed", (3) the issue has to do with the WebAssembly runtime (i.e. the issue is classified in the *WebAssembly* component of the V8 engine), and then the $BugSet1_{V8}$ is obtained directly through the download link provided by the issue tracking system.

Collecting SpiderMonkey Bugs. In the issue tracking system of SpiderMonkey[3], we first apply the following rule filters to select WebAssembly runtime bugs: (1) the issue is of the type "DEFECT", (2) its status is "CLOSED", "RESOLVED", or "VERIFIED", (3) its "RESOLUTION" field is assigned to "FIXED", (4) the issue has to do with the WebAssembly runtime (i.e. the issue is classified in the *Javascript: WebAssembly* component of the SpiderMonkey engine), and then the filtered bug list is downloaded directly through the link provided by the issue tracking system to get $BugSet1_{SpiderMonkey}$.

Collecting Wasmer/Wasmtime Bugs. Both Wasmer and Wasmtime use GitHub as their issue tracking system[4,5], where developers categorize reported issues by giving each issue a different tag. To collect as much and complete bug data as possible, we refer to the method of the paper [24, 26-28] to include Wasmer and Wasmtime closed issues and pull requests into the screening scope. We build two queries to get the issues related to the WebAssembly runtime. Specifically, for Wasmer, we look for closed issues and pull requests whose label is assigned: "bug". For Wasmtime, we focus on closed issues and pull requests in GitHub repositories that have at least one label assigned: "bug" or "fuzz-bug". We used the GitHub REST API [24] to build the crawler and retrieved 403 bugs for Wasmer and 165 bugs for Wasmtime, respectively. But pull requests are a kind of special issue with source code fix information, which sometimes exists directly as a bug-fixing commit of another issue [25], and sometimes exists as a separate issue. So in Wasmer and Wasmtime, when a pull request as a bug-fixing commit exists, it is necessary to determine whether the issue that the pull request links to already exists. And if the issue already exists, it is considered a duplicate bug that must be filtered. Finally, we get $BugSet1_{Wasmer}$ containing 380 bugs and $BugSet1_{Wasmtime}$ having 165 bugs.

TABLE I
THE STATISTICS OF THE SUBJECTS COLLECTED IN OUR STUDY.

Runtime	Duration Time	BugSet1	BugSet2	BugSet3
V8	2016-08-23-2022-2-18	477	220	196
SpiderMonkey	2018-06-10-2022-03-31	453	325	325
Wasmer	2019-02-16-2022-03-15	380	224	209
Wasmtime	2019-12-05-2022-04-11	165	98	88
Total	-	1,475	867	818

2) Collection of Bug-Fixing Commits: We then use the link that exists between the bug report and the bug-fixing commit to identify the fix information for bugs in $BugSet1$. To reduce the noise of the data, we only focus on the case where a bug corresponds to only one bug-fixing commit, and filter out the bug data where a bug corresponds to multiple bug-fixing commits or a bug has no corresponding bug-fixing commit [21], [26]. In addition, for the few cases where a bug-fixing commit fixes multiple bugs when classifying root causes and symptoms of bugs, we follow the existing work [27]–[29] and classify bugs and bug-fixing commits one by one as separate individuals to obtain $BugSet2'$. But when analyzing bugs for the fix information, to reduce noise, such bug data is filtered out to obtain $BugSet3'$.

In this study, we only focus on coding-related bug fixes [22], [23], [26], [30]. Therefore, $BugSet2'$ and $BugSet3'$ filter out bugs whose fixes only involve unrelated files (e.g., test cases, documents, change logs, *.md, .gitignore, LICENSE) and unrelated changes (e.g., only change comments that are unrelated to bug-fixing code). Finally, the research datasets $BugSet2$ and $BugSet3$ for this experiment are obtained, as shown in Table I.

C. Bug Classification and Labeling

To characterize WebAssembly runtime bugs, we labeled the root cause and symptom of each WebAssembly runtime bug in $BugSet2$. Our classification of WebAssembly runtime bug root causes and symptoms refers to the definitions of bug root causes and symptoms in existing studies [27]–[29], [31]–[34]. For each bug, we examined its issue messages, comments,

[2]https://bugs.chromium.org/p/v8/issues/list?q=&can=1
[3]https://bugzilla.mozilla.org/query.cgi?format=advanced
[4]https://github.com/wasmerio/wasmer
[5]https://github.com/bytecodealliance/wasmtime

commit messages, and linked pull request messages to determine the root cause and symptom categories.

To minimize subjectivity bias in the labeling process, the labeling of root causes and symptoms of WebAssembly runtime bugs was performed manually by three authors familiar with the WebAssembly runtime projects. First, the bugs were randomly divided into 10 sets (the first 9 containing 87 bugs and the last containing 84 bugs). Then two authors independently labeled the root causes and symptoms of each set of bugs. After each set of bugs was labeled, the two authors cross-checked the label results and discussed the conflicting labels until they reached an agreement (the process was repeated 10 times). During the repeated iterations of the discussion, the two authors could be more clear about the categories of the current bug labels. Finally, the third author verified the label results of all bugs and worked with the original two authors to resolve any disagreements.

D. Root Causes of WebAssembly Runtime Bugs

Referring to the classification of root causes in existing work [27]–[29], [31]–[34], and then after the mentioned process of classification and labeling (Section III-C), we conclude the following 16 root causes of WebAssembly runtime bugs.

- **Incorrect Algorithm Implementation:** This root cause is related to errors in the set of code steps that implement the solution to a specific issue or calculation. Its fix is usually found in function/method definitions that contain some statements or blocks with a logical structure.
- **Memory:** This situation involves incorrect memory handling, such as incorrect/failed memory allocation, memory leaks, dangling pointers, illegal accesses, etc.
- **Incorrect Exception Handling:** This type of bug is caused by incorrect exception handling, e.g., incorrect or inaccurate error information thrown, missing/redundant exceptions, wrong class of exceptions thrown, etc.
- **API Misuse:** This type of bug can be subdivided into two subcategories in WebAssembly runtime: 1) *API Missing/Redundancy:* This is because the developer is missing /redundant the use of an API in the code; 2) *Wrong API:* It is caused by the developer using a wrong API name, argument, or receiver, which violates the API usage restrictions.
- **Type Problem:** These bugs are caused by type-related issues, e.g. type conversion, type checking, and type inference.
- **Incorrect Condition Logic:** This is caused by an incorrect conditional expression. This type of bug is usually fixed in statement blocks such as if-else, for loops, etc.
- **Incorrect Assignment:** This is caused by variables that have been incorrectly initialized, assigned, or not initialized.
- **Incorrect Configuration:** These bugs are related to incorrect configuration of compilation, build, compatibility, and installation files. The result of these bugs will either fail to build or behave unexpectedly.
- **Missing Condition Checks:** This type of bug lacks the necessary conditional statements to handle special cases, such as boundary values, null checks, etc.
- **Concurrency:** These bugs relate to incorrect manipulation of concurrent oriented structures (e.g., threads, locks, shared memory, and race conditions).
- **Environment Incompatibility:** These bugs are related to not correctly handling some characteristics (e.g. the number of bytes of the architecture) of a particular environment, for example, hardware or operating system.
- **External API Incompatibility:** This type of bug is caused by an API incompatibility between the WebAssembly runtime and a third-party library, which is usually associated with updates to the third-party library.
- **Dependent Module Issue:** These bugs are brought by failing to import necessary dependent modules or by importing the incorrect modules.
- **Logical Order Error:** These bugs are caused by an illogical ordering of the statements, and the solution is usually to rearrange the statements.
- **Incorrect Numerical Computation:** This type of bug is caused by incorrect use of operands or operators, missing operands in calculations, and other incorrect numerical calculations.
- **Others:** This type of bug cannot be assigned to any of the above categories and occurs very infrequently.

E. Symptoms of WebAssembly Runtime Bugs

Referring to the classification of symptoms in existing work [27]–[29], [33], [34], according to the classification and labeling process (Section III-C) described above, we summarize the following 6 symptoms for WebAssembly runtime bugs.

- **Crash:** This symptom indicates that the WebAssembly runtime has abruptly terminated, which is typically accompanied by an error message.
- **Incorrect Functionality:** This symptom arises when the WebAssembly runtime operates improperly but does not crash, for example, by yielding unexpected results or an inaccurate intermediate state.
- **Build Error:** This symptom means incorrect compilation, build, and installation of a WebAssembly runtime.
- **Bad Performance:** This symptom denotes using up more time or resources than anticipated (e.g. memory).
- **Hang:** This symptom means that the WebAssembly runtime is still not responding after running for a long time.
- **Unreported:** The symptoms of WebAssembly runtime bugs cannot be determined by looking through the issue messages, comments, commit messages, and linked pull request messages.

IV. RESULTS AND ANALYSIS

In this section, we discuss and analyze the experimental results obtained according to the method described above.

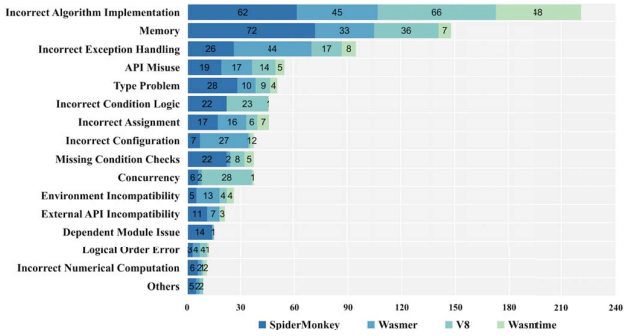


Fig. 2. The distribution of bugs by root causes.

A. RQ1: How are the root causes of WebAssembly runtime bugs distributed?

Fig. 2 shows the distribution of WebAssembly runtime bugs by root causes. As demonstrated by the figure, the most frequent root cause, accounting for 221 bugs, is *Incorrect Algorithm Implementation*, which includes 62 in SpiderMonkey, 45 in Wasmer, 66 in V8, and 48 in Wasmtime. Among them, the bugs of *Incorrect Algorithm Implementation* in Wasmtime account for 48.98% of all its bugs, which is the highest percentage among the four WebAssembly runtimes. An observation of this part of bugs finds that *Incorrect Algorithm Implementation* in Wasmtime always occurs in the code generation phase, which deserves developers of Wasmtime to devote more attention.

Finding #1: *Incorrect Algorithm Implementation accounts for 25.49% of WebAssembly runtime bugs, the most prevalent of all root causes.*

The second most prevalent root cause, *Memory*, is responsible for 148 bugs, including 72 in SpiderMonkey, 33 in Wasmer, 36 in V8, and 7 in Wasmtime. This phenomenon should be related to the direct access to raw bytes and manual memory management of WebAssembly. WebAssembly runtime memory is an abstract RAM (essentially a linear array of bytes) with only the most basic read and writes functionality, very close to the bottom, with no advanced memory management and garbage collection implemented. The developers are required to manage memory, and segment errors or memory leaks can easily occur if memory is not used properly. This result shows that safe handling of memory at the WebAssembly runtime is quite difficult and requires further research by the WebAssembly runtime developers.

Finding #2: *Memory leads to a large number of WebAssembly runtime bugs, accounting for 17.07% of all bugs. This result indicates that memory safety management in WebAssembly runtime is challenging.*

According to Fig. 2, *Incorrect Exception Handling* has 26 in SpiderMonkey, 44 in Wasmer, 17 in V8, and 8 in Wasmtime, which is the third-ranked root cause. The reason the WebAssembly runtime contains many incorrect exceptions handling bugs is probably due to the exception interaction between the WebAssembly runtime and the embedder. In the exception handling of WebAssembly runtime, an exception to some instructions will generate a trap, which will immediately abort the current computation. But traps can be handled without the WebAssembly runtime, because an embedder usually provides a way to handle such cases, for example, by specifying them as JavaScript exceptions.

TABLE II
THE DISTRIBUTION OF API MISUSE BUGS.

Runtime	API M/R	Wrong API			
		Receiver	Name	Argument	Total
SpiderMonkey	5	0	7	7	14
Wasmer	3	1	4	9	14
V8	6	0	1	7	8
Wasmtime	2	1	1	1	3
Total	16	2	13	24	39

M/R is an abbreviation for Missing/Redundancy.

Finding #3: *Incorrect Exception Handling accounts for 10.96% of WebAssembly runtime bugs, the third-ranked root cause.*

API Misuse is the fourth largest source of WebAssembly runtime bugs. We refer to existing work [27], [28], [35], [36] to subdivide such bugs into two subcategories. Table II shows the results, *Wrong API* is the most prevalent subcategory leading to *API Misuse* in WebAssembly runtime, accounting for 70.91%, followed by *API Missing/Redundancy*, accounting for 29.09%. Among the *Wrong API*, wrong API parameters account for the largest proportion (61.54%), followed by wrong API names (33.33%). This indicates on the one hand that developers may lack domain knowledge related to API usage, and on the other hand, the need for better and more detailed documentation of API usage by developers. In addition, *API Missing/Redundancy* is the subclass that most frequently leads to *API Misuse* in MuBench [35], [37] (one of the most extensive benchmarks for *API Misuse* studies, which researches 90 actual *API Misuse* bugs from multiple Java projects). This differs from the *API Misuse* subcategories distribution in the WebAssembly runtime. On the one hand, existing API misuse detectors are more likely to detect bugs caused by *API Missing/Redundancy* [35], so it is necessary to develop an API misuse detection technique that is distinct from conventional software for the WebAssembly runtime. On the other hand, existing API misuse detectors have limitations in terms of precision and recall [35], [38], so our bug data can be used to train existing API misuse detectors to help improve their functionality.

Finding #4: *API Misuse is the fourth-ranked root cause, with Wrong API being the most likely subcategory to cause API Misuse at 70.91%.*

B. RQ2: How are the symptoms of WebAssembly runtime bugs distributed?

Fig. 3 shows the distribution of bugs by symptoms. The most frequent symptom is discovered to be *Crash*, which accounts for 493 bugs, including 180 in SpiderMonkey, 117 in

TABLE III
THE DISTRIBUTION OF BUGS IN EACH ROOT CAUSE CATEGORY BY SYMPTOMS.

Root Cause \ Symptom	Crash	Incorrect Functionality	Build Error	Bad Performance	Hang	Unreported	$Total_{symptom}$
Incorrect Algorithm Implementation	103	69	16	21	4	8	221
Memory	122	16	1	4	1	4	148
Incorrect Exception Handling	78	14	0	2	1	0	95
API Misuse	25	24	5	1	0	0	55
Type Problem	28	15	4	0	0	4	51
Incorrect Assignment	17	16	9	1	0	3	46
Incorrect Condition Logic	30	14	1	1	0	0	46
Concurrency	28	5	0	0	2	2	37
Incorrect Configuration	3	4	28	0	0	2	37
Missing Condition Checks	24	3	5	2	1	2	37
Environment Incompatibility	9	0	17	0	0	0	26
External API Incompatibility	10	0	9	0	0	2	21
Dependent Module Issue	0	1	14	0	0	0	15
Logical Order Error	7	4	0	0	0	1	12
Incorrect Numerical Computation	6	5	0	0	0	0	11
Others	3	2	3	1	0	0	9
$Total_{cause}$	493	192	112	33	9	28	867

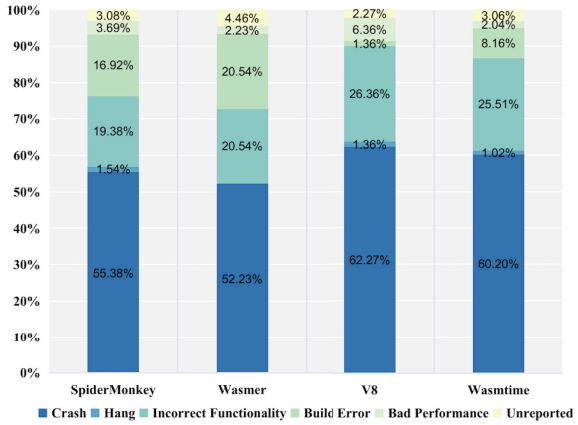


Fig. 3. The distribution of bugs by symptoms.

Wasmer, 137 in V8, and 59 in Wasmtime. This shows that if the developers of WebAssembly runtime cannot discover and deal with such bugs in time, the user experience will be greatly reduced. Therefore, the introduction of exception-handling strategies can be considered to improve the robustness of the WebAssembly runtime. In addition, when the WebAssembly runtime crashes, it often carries error message reports which can help developers locate and debug such bugs.

Finding #5: *The most prevalent symptom is Crash, which accounts for 56.86% of WebAssembly runtime bugs.*

Incorrect Functionality is the second most prevalent symptom, with a total of 192 bugs, including 63 in SpiderMonkey, 46 in Wasmer, 58 in V8, and 25 in Wasmtime. This symptom is considered application-specific [33], so a test oracle setup to detect such bugs requires developers to have domain-specific knowledge, which creates many challenges for testing and fixing bugs that exhibit such symptoms.

Finding #6: *Incorrect Functionality accounts for 22.15% of WebAssembly runtime bugs, and testing and fixing such bugs requires developers to have domain-specific knowledge.*

Fig. 3 illustrates that in the symptom category of WebAssembly runtime bugs, *Build Error* ranks third, accounting for 12.92%. Of these, bugs that manifest as *Build Error* account for more in SpiderMonkey (16.92%) and Wasmer (20.54%) than in V8 (1.36%) and Wasmtime (8.16%). This demonstrates that compiling, building, and installing SpiderMonkey and Wasmer is challenging and deserve more attention from developers. Other than that, the symptoms of *Bad Performance* and *Hang* account for just 3.81% and 1.04%, respectively, of WebAssembly runtime bugs.

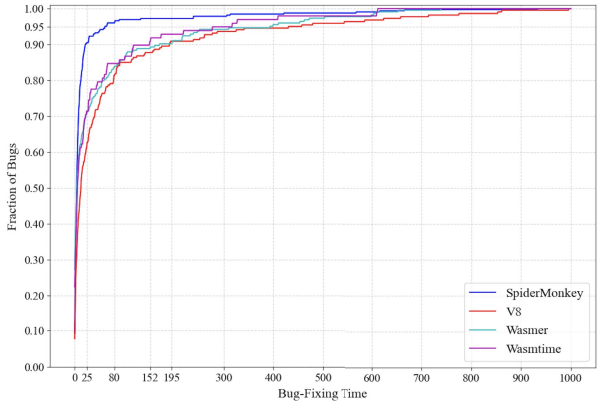
Finding #7: *Build Error ranks third in the symptoms, accounting for 12.92% of WebAssembly runtime bugs.*

C. RQ3: What is the connection between WebAssembly runtime bug root causes and symptoms?

Table III shows the number of symptoms exhibited by the bugs in each root cause category. As can be seen, the top three root causes, *Incorrect Algorithm Implementation*, *Memory*, and *Incorrect Exception Handling*, occur in practically all symptom categories, except for the *Build Error* and *Unreported* categories in *Incorrect Exception Handling*. Bugs caused by these three root causes account for 53.52% of all bugs, which indicates that these bugs occur frequently and have various impacts. Therefore, WebAssembly runtime developers ought to give the creation of tools for detecting, locating, and fixing such bugs more consideration.

Finding #8: *Incorrect Algorithm Implementation, Memory, and Incorrect Exception Handling can lead to almost all types of bug symptoms. They account for 53.52% of all bugs.*

According to Table III, the two most frequent symptoms of WebAssembly runtime are *Crash* and *Incorrect Functionality*. Except for *Dependent Module Issue* in *Crash*, *External API Incompatibility* and *Environment Incompatibility* in *Incorrect Functionality*, all other categories of root causes can induce these two symptoms. This suggests that developers should pay more attention to bug detection for these two symptoms, such

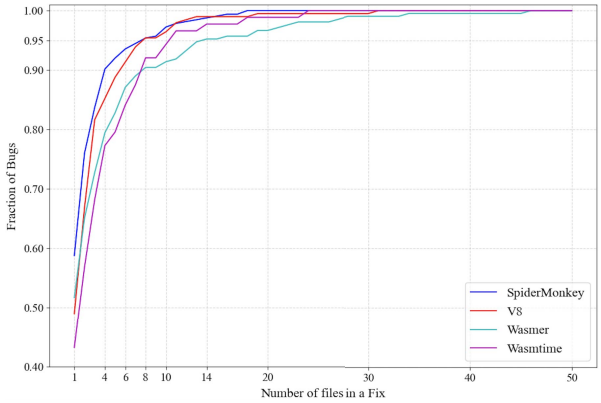


(a) The empirical cumulative distribution function of bug-fixing time.

Runtime	Mean	Median	SD	Min	Max
V8	71.88	13	160.62	0	995
SpiderMonkey	21.78	4	86.82	0	933
Wasmer	55.06	5	126.97	0	739
Wasmtime	48.29	6	110.60	0	612

(b) The statistics of the bug-fixing time.

Fig. 4. The bug-fixing time for V8, SpiderMonkey, Wasmer, and Wasmtime in bug fixes.



(a) The empirical cumulative distribution of the number of files involved in a bug fix.

Runtime	Mean	Median	SD	Min	Max
V8	2.72	2	3.32	1	31
SpiderMonkey	2.32	1	2.70	1	18
Wasmer	3.75	1	5.89	1	46
Wasmtime	3.51	2	3.95	1	24

(b) The statistics on the number of files involved in bug fixes.

Fig. 5. The number of files involved in bug fixes.

as designing effective test oracles, which can find WebAssembly runtime bugs caused by multiple root causes.

Finding #9: *WebAssembly runtime with 79.01% of the consequences of bugs are Crash and Incorrect Functionality, and the root causes of these two symptoms are varied.*

Bad Performance is only strongly associated with *Incorrect Algorithm Implementation*. In particular, 63.64% of the 33 bugs demonstrating *Bad Performance* are brought on by *Incorrect Algorithm Implementation*. *Build Error* is strongly associated with multiple root causes, with *Incorrect Configuration* having the highest number of 28, followed by *Environment Incompatibility* with 17, *Incorrect Algorithm Implementation* with 16, and *Dependent Module Issue* with 14. This clear correlation can help developers speed up the debugging process. When a bug is one of these two symptoms, developers can quickly investigate the root cause that is highly correlated with the symptom to narrow down the suspects.

Finding #10: *Bad Performance is closely associated with Incorrect Algorithm Implementation. Build Error is closely associated with four root causes (i.e. Incorrect Configuration, Environment Incompatibility, Incorrect Algorithm Implementation, and Dependent Module Issue).*

D. RQ4: How long do WebAssembly runtime bugs take to fix?

The distribution of bug-fixing across time is examined by this research question. In this study, bug-fixing time is defined as the time interval between the date when a bug report is filed in the bug tracking system and the date when the bug is fixed and will not be changed again. The ideal bug-fixing time is 0 days, which means that a bug is fixed as soon as it is reported. But in practice, it usually takes longer to fix a bug, sometimes even years, because of resource and time constraints, etc. We collect the start time and end time of bugs in $BugSet2$ by referring to existing work [22], [23], and calculate their time interval, i.e., bug-fixing time, in days.

The relationship between the bug-fixing time and the percentage of bugs is depicted in Fig. 4(a). It can be seen that about 90% of the bugs in V8, SpiderMonkey, Wasmer, and Wasmtime are fixed within 195, 25, 195, and 152 days, respectively. Fig. 4(b) provides statistics on bug-fixing time (including mean, median, standard deviation (SD), minimum (Min), and maximum (Max)). As can be observed, for V8, SpiderMonkey, Wasmer, and Wasmtime, the median bug-fixing time are 13, 5, and 6 days, respectively. On average, bugs are repaired in 71.88 days for V8, 21.78 days for SpiderMonkey, 55.06 days for Wasmer, and 48.29 days for Wasmtime. The bug-fixing time of SpiderMonkey is much shorter than V8, Wasmer, and Wasmtime, which indicates that the developers of SpiderMonkey are more active in bug maintenance. Moreover, the bug-fixing time for these four WebAssembly runtimes is lower than the average fixing time of 111 and 98 days for the popular compilers GCC and LLVM in the existing study [22].

Finding #11: *At the median, the bug-fixing time are 13, 4, 5, and 6 days for V8, SpiderMonkey, Wasmer, and Wasmtime respectively.*

E. RQ5: How many files and lines of code must be changed to fix a WebAssembly runtime bug?

This research question investigates the number of files and lines of code involved in fixing bugs in V8, SpiderMonkey, Wasmer, and Wasmtime. A deeper comprehension of bug fixes in the WebAssembly runtime can provide useful guid-

TABLE IV
THE STATISTICS ON THE NUMBER OF FILES INVOLVED IN BUG FIXES FOR EACH ROOT CAUSE CATEGORY.

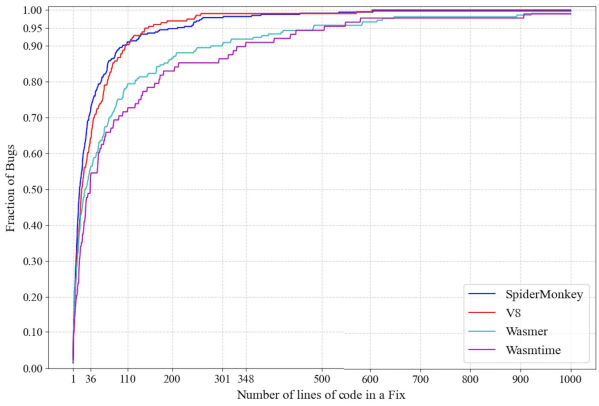
Root causes	Mean	Median	SD	Min	Max
Incorrect Algorithm Implementation	3.81	2.5	3.65	1	19
Memory	3.34	2	4.31	1	27
Incorrect Exception Handling	3.24	1	4.25	1	28
API Misuse	1.16	1	0.46	1	3
Type Problem	2.47	1	2.34	1	10
Incorrect Assignment	1.39	1	1.42	1	10
Incorrect Condition Logic	1.43	1	1.04	1	6
Concurrency	4.34	2	8.71	1	46
Incorrect Configuration	2.83	1	6.04	1	34
Missing Condition Checks	1.22	1	0.48	1	3
Environment Incompatibility	4.13	2	6.62	1	31
External API Incompatibility	5.84	3	10.05	1	46
Dependent Module Issue	1.40	1	0.83	1	4
Logical Order Error	1.00	1	0	1	1
Incorrect Numerical Computation	1.36	1	0.92	1	4
Others	4.00	1	7.19	1	38

TABLE V
THE STATISTICS ON THE NUMBER OF FILES INVOLVED IN BUG FIXES FOR EACH SYMPTOM CATEGORY.

Symptoms	Mean	Median	SD	Min	Max
Crash	2.87	1	3.76	1	31
Incorrect Functionality	2.77	1	4.44	1	46
Build Error	2.52	1	3.27	1	23
Bad Performance	4.45	2	4.51	1	19
Hang	1.22	1	0.44	1	2
Unreported	4.65	2	7.26	1	34

ance on automated debugging techniques for WebAssembly runtime bugs, in addition to helping developers design better WebAssembly runtimes. We use the bug-fixing commits of $BugSet3$ as the object of our study, and extract information about files and lines of code related to source code changes by parsing the patches in the bug-fixing commits. Specifically, we do not consider changes in unrelated files (e.g., test cases, documents, change logs, *.md, .gitignore, LICENSE) [22], [23], [26].

*1) **Number of files**:* Fig. 5(a) depicts the empirical cumulative distribution of the number of files necessary to fix a bug. From Fig. 5(a), it can be seen that about 90% of the bug fixes in SpiderMonkey involve 4 files, while more than 90% of the bug fixes in V8, Wasmer, and Wasmtime involve no more than 6, 8, and 8 files, respectively. In particular, 48.98%, 58.77%, 51.67%, and 43.18% of the bug fixes in V8, SpiderMonkey, Wasmer, and Wasmtime, respectively, involve only one file. In addition, most bug fixes in the four WebAssembly runtimes involve no more than 30 files at most. Fig. 5(b) provides statistics on the number of files in bug fixes (including mean, median, standard deviation (SD), minimum (Min), and maximum (Max)). On average, V8 and SpiderMonkey require 2.72 and 2.32 file modifications to fix a bug, respectively, while Wasmer and Wasmtime require 3.75 and 3.51 files, respectively. This may indicate that the code implementations of V8 and SpiderMonkey have a better modular design compared to Wasmer and Wasmtime since the bug fixes in V8 and SpiderMonkey involve fewer files than in Wasmer and Wasmtime.



(a) The empirical cumulative distribution of lines of code involved in a bug fix.

Runtime	Mean	Median	SD	Min	Max
V8	45.40	18.50	74.04	1	604
SpiderMonkey	45.42	14	98.09	1	1,167
Wasmer	113.19	26	307.08	1	3,661
Wasmtime	114.95	36	189.09	1	1,036

(b) The statistics for the lines of code involved in bug fixes.

Fig. 6. The lines of code involved in bug fixes.

Finding #12: *Over 50% of bug fixes in the four WebAssembly runtimes involve only one file, while fixes for more than 90% of bugs involve no more than 8 files.*

Table IV shows statistics on the files involved in bug fixes for each root cause category (including mean, median, standard deviation (SD), minimum (Min), and maximum (Max)). From Table IV, *External API Incompatibility*, *Concurrency*, *Environment Incompatibility*, and *Incorrect Algorithm Implementation* are the four root causes for the highest number of files involved in bugs fixes (except for the *Others* category), with a mean of 5.84, 4.34, 4.13, and 3.81, and a median of 3, 2, 2, and 2.5, respectively. Furthermore, *Incorrect Algorithm Implementation* has the highest percentage of bugs at 25.49%, which indicates that this category has a greater weight and developers should pay more attention.

Finding #13: *The bug fixes for External API Incompatibility, Concurrency, Environment Incompatibility, and Incorrect Algorithm Implementation rank the top four (except Others) in terms of the number of files involved, with an average of 5.84, 4.34, 4.13, and 3.81.*

Table V provides statistics for the files involved in bug fixes for each symptom category. On average, *Bad Performance* has far more files involved in bug fixes than the other symptom categories (except *Unreported*), but it only accounts for 3.81% of all bugs. *Crash* (except *Unreported*) is next, with its bug fixes involving an average of 2.87 files and exhibiting *Crash* bugs accounting for 56.86% of all bugs. *Hang* is the symptom involving the fewest files (except *Unreported*), with an average of only 1.22 files. This suggests that the *Crash* symptom deserves more attention from developers.

TABLE VI
THE STATISTICS FOR THE NUMBER OF LINES OF CODE INVOLVED IN BUG FIXES FOR EACH ROOT CAUSE CATEGORY.

Root causes	Mean	Median	SD	Min	Max
Incorrect Algorithm Implementation	103.82	58.5	128.96	3	907
Memory	108.34	29	347.15	1	3,661
Incorrect Exception Handling	75.07	28	133.16	2	893
API Misuse	5.93	3	8.35	1	52
Type Problem	43.04	24	63.97	2	363
Incorrect Assignment	11.8	5	22.45	1	138
Incorrect Condition Logic	14.27	9	14.79	2	68
Concurrency	124.31	31	319.01	1	1,551
Incorrect Configuration	20.44	8.5	34.28	1	174
Missing Condition Checks	12.92	8.5	14.72	1	70
Environment Incompatibility	114.96	28.5	192.9	4	604
External API Incompatibility	65	20	145.46	1	648
Dependent Module Issue	36.6	6	75.67	1	259
Logical Order Error	11.13	11	5.17	2	17
Incorrect Numerical Computation	5.73	2	5.53	2	18
Others	92.67	11	14.72	1	485

TABLE VII
THE STATISTICS FOR THE NUMBER OF LINES OF CODE INVOLVED IN BUG FIXES FOR EACH SYMPTOM CATEGORY.

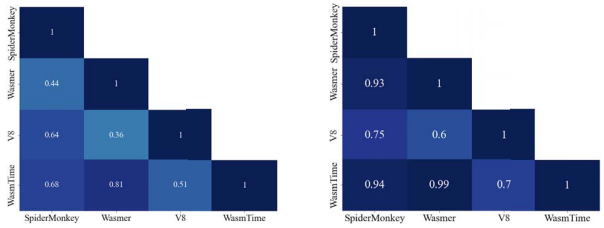
Symptoms	Mean	Median	SD	Min	Max
Crash	74.80	21	215.19	1	3,661
Incorrect Functionality	63.6	16	151.61	1	1,551
Build Error	47.78	12	103.89	1	585
Bad Performance	110.06	70	113.15	2	394
Hang	45	23	47.32	4	129
Unreported	86.19	16.5	159.36	1	606

Finding #14: *Bug fixes for Bad Performance involve the highest number of files (except Unreported), with an average of 4.45. Bug fixes for Crash have the second highest number of files (except Unreported), with an average of 2.87.*

2) ***Lines of Code****:* We investigate changes in the source code lines related to bug fixes in addition to the files. In our study, the number of modified lines of code is defined as the sum of the number of lines of code added and the number of lines of code subtracted. As shown in Fig. 6(a), over 50% of the bug fixes in the four WebAssembly runtimes involve less than 36 lines of source code. About 90% of the bug fixes in V8, SpiderMonkey, Wasmer, and Wasmtime involve no more than 110, 110, 301, and 348 lines of code, respectively. As can be seen in Fig. 6(b), the means of the lines of code to fix a bug in V8, SpiderMonkey, Wasmer, and Wasmtime are 45.40, 45.42, 113.19, and 114.95, with the median lines of code being 18.5, 14, 26, and 36. This suggests that bugs in V8 and SpiderMonkey are usually localized, and despite their overall intricacy, the codebase is little affected overall by the bug fixes.

Finding #15: *The median source code lines for bug fixes for V8, SpiderMonkey, Wasmer, and Wasmtime are 18.5, 14, 26, and 36 lines, respectively.*

The statistics on the lines of code used to fix bugs for each root cause category are shown in Table VI (including mean, median, standard deviation (SD), minimum (Min), and maximum (Max)). In terms of median, bug fixes for *Incorrect Algorithm Implementation* require the most lines of code involving 58.5 lines with a mean of 103.82, which indicates that bugs in this root cause category may have high challenges in locating



(a) The correlation of root causes. (b) The correlation of symptoms.

Fig. 7. The correlation between WebAssembly runtimes.

bugs and fixing them automatically [39]–[41]. In terms of average, the bug fixes of *API Misuse*, *Incorrect Assignment*, *Incorrect Condition Logic*, *Missing Condition Checks*, *Logical Order Error*, and *Incorrect Numerical Computation* all involve less than 20 lines of code, and the bugs caused by these six root causes account for 23.88% of all bugs, slightly less than the 25.49% of *Incorrect Algorithm Implementation*.

Finding #16: *At the median, Incorrect Algorithm Implementation bug fixes require the most lines of code involved at 58.5 lines, with a mean of 103.82. Bug fixes for the six categories of root cause involved an average of fewer than 20 lines of code, representing 23.88% of all bugs.*

Table VII provides statistics for the number of lines of code involved in bug fixes for each symptom category. *Bad Performance* bug fixes involve the highest number of lines of code with an average of 110.06 lines and a median of 70 lines. This may be due to the strong correlation between *Bad Performance* and *Incorrect Algorithm Implementation*. On average, the second highest number of lines of code involved in bug fixes was *Crash* with 74.80 lines, followed by *Incorrect Functionality* with 63.60 lines (except *Unreported*). Similar to the files involved, *Hang* is the symptom involving the least number of lines of repair code, with an average of only 45 lines.

Finding #17: *On average, the bug fixes of Bad Performance involve the most lines of code with 110.06 lines. Hang is the symptom that involves the least number of lines of code fixed, with an average of only 45 lines.*

F. RQ6: Do the bugs of different WebAssembly runtimes have anything in common?

To measure the commonality between WebAssembly runtimes, referring to existing work [27], [28], [42], we computed Spearman correlations for each pair of WebAssembly runtimes based on their root cause distribution and symptom distribution. The Spearman correlation coefficient is a statistical measure used to indicate the strength of a monotonic relationship between two pairwise variables [43]. Fig. 7 displays the results of the correlations, where [0.8, 1.0] indicates a very strong correlation, [0.6, 0.79] indicates a strong correlation, [0.4, 0.59] indicates a moderate correlation, and [0.2, 0.39] is a weak correlation.

In terms of the root causes, Fig. 7(a) shows that there is a very strong correlation between Wasmer and Wasmtime, and a strong correlation between SpiderMonkey and V8, Wasmtime. However, there are weak, moderate correlations between V8 and Wasmer, Wasmtime. From Fig. 2, it can be seen that this may be related to the two root causes of *Incorrect Condition Logic*, and *Concurrency*. They are more distributed in V8, accounting for 10.45%, and 12.73% of V8, but very little in Wasmer and Wasmtime, respectively. This suggests that developers should distinguish between differences and consistency in the root cause of bugs across different WebAssembly runtimes when designing solutions for locating and fixing WebAssembly runtime bugs.

From Fig. 7(b), it can be found that the four WebAssembly runtimes have a strong correlation in the symptoms, which indicates the generality of our findings on the symptoms exhibited by bugs. It also further demonstrates the possibility of creating common tests, debugging, and repair techniques for different WebAssembly runtimes.

Finding #18: *The four WebAssembly runtimes have moderate and strong correlations in root causes of bugs (except for V8 and Wasmer) and strong correlations in symptoms of bugs.*

V. Discussion

We will analyze the broader implications of our findings using the important findings from the preceding section. In particular, lessons will be learned from the findings to direct future work on testing, debugging, identifying, and resolving WebAssembly runtime bugs.

Findings 1, 2, 3, and 8 show that bugs in *Incorrect Algorithm Implementation*, *Memory*, and *Incorrect Exception Handling* occur frequently and can cause almost every symptom. Therefore, designing techniques that can automatically detect and locate bugs caused by these three root causes is a promising research direction. Furthermore, according to finding 16, the fixing of bugs caused by *Incorrect Algorithm Implementation* often involves multiple lines of code, with an average of 103.82 lines of code. This suggests that existing bug-fixing techniques may need to be enhanced to address multi-point bugs, i.e., bugs that exist in multiple statements.

The two symptoms of WebAssembly runtime bugs that account for the majority of bugs (79%) are *Crash* and *Incorrect Functionality* (Findings 5, 6, and 9). Therefore, it is necessary to design automatic testing techniques for them, which can facilitate WebAssembly runtime to detect bugs caused by various root causes. *Crash* bugs usually have well-defined test oracles, so automated test input generation techniques may be a promising option for testing Crash bugs. Since *Incorrect Functionality* bugs involve determining the correctness of the intermediate state of the program, their test oracles are difficult to define. Differential testing [40] might be a potential direction. In addition, bugs that exhibit *Crash* often come with an error message, which can help developers locate and debug bugs. For example, Wu et al [44] proposed a method to locate buggy classes and functions using error messages carried by bugs that exhibit crashes.

Findings 12, 13, and 14 can provide useful information for studying WebAssembly runtime automatic location and debugging techniques. For example, our findings show that the average number of files involved in the fix for a *Crash* bug is 2.87, which can help developers narrow down suspicious files when trying to isolate bugs that exhibit Crash.

The fixes for bugs caused by six categories of root causes at WebAssembly runtime involve an average of fewer than 20 lines of code and account for 23.88% of all bugs (Finding 16). And these six categories of root causes (i.e., *API Misuse*, *Incorrect Assignment*, *Incorrect Condition Logic*, *Missing Condition Checks*, *Logical Order Error*, and *Incorrect Numerical Computation*) can cause all bug symptoms (Table III). This suggests that many bugs in the WebAssembly runtime are amenable to existing techniques for automatic bug detection, localization, and repair [41], which could aid in the development of these techniques.

VI. Threats to validity

Internal threats. The classification for bugs may be inaccurate, which poses an internal threat to the validity of our results. To mitigate this threat, we refer to existing work [27]–[29], [31] for classification definitions of bug root causes and symptoms to initiate our labeling process. To minimize subjectivity bias in the labeling process, the classification of WebAssembly runtime bugs was done manually by three authors who are familiar with WebAssembly runtime projects. During the classification process, the two authors first label the bug data independently, then cross-check the label results of each other and discuss the differences until a consensus is reached. Finally, the third author verifies all the labeled data, and for those data that still conflict, the three authors continue the discussion until a consensus is reached. Despite the great care taken by the authors, there is still the possibility of incorrect classification, which cannot be completely avoided.

External Threats. The representativeness of the bug dataset used in this study may be a threat to external validity. To reduce this threat, we systematically collected 867 bugs from four popular WebAssembly runtimes as our study data. To make sure we focus on the true bugs and bug fixes, we first chose only closed and fixed issues that are marked as "bugs" [21]–[23], [26], [30], and then we collected fixes for bugs as per existing work [21]–[23], [26]. According to the results of the commonality analysis in Section IV, the root causes of bugs are mostly moderately and strongly correlated, while the symptoms are strongly correlated across the four WebAssembly runtimes. This indicates that our findings are representative and general.

VII. Related Work

Empirical Studies on Software Bugs. The most relevant to our study is the qualitative and quantitative study by Romano et al. [45] on bugs in three popular WebAssembly compilers. Our study differs fundamentally from [45] due to the different research tools. The WebAssembly compiler they studied is a tool for compiling source programs written in

high-level languages (e.g., C/C++/Rust) into WebAssembly binary modules. However, the WebAssembly runtime we study is the program responsible for parsing and executing the WebAssembly binary module. As far as we know, we are the first to perform an empirical analysis of the bug characteristics of the WebAssembly runtime.

Besides, many research works have been done in studying bugs in software systems [20]–[23], [27]–[30], [42], [45]–[49]. For example, Shen et al. [28] conducted an empirical study on the bugs of deep learning compilers (TVM, Glow, and nGraph). Garcia et al. [29] carried out a study for autonomous vehicle software systems bugs. Ocariza et al. [47] investigated client-side JavaScript bug characteristics.

WebAssembly Runtime Development. WebAssembly runtimes have been developed for a variety of domains [2], [4], [7]–[11], [16], [17], [50]–[52]. For example, Wen et al. [8] developed an operating system, which enables IoT and edge devices to safely and efficiently run WebAssembly applications. Salim et al. [11] developed TruffleWasm to execute standalone WebAssembly modules while also providing interoperability with other GraalVM-hosted languages. For IoT devices with constrained resources, Jacobsson et al. [20] implemented a WebAssembly interpreter. Our research focuses on the bug characteristics of the WebAssembly runtime, which can provide an auxiliary role in the development of WebAssembly runtime research applicable to various domains.

VIII. Conclusion

As the WebAssembly runtime is increasingly developed and used in various domains, the quality of the WebAssembly runtime is becoming more and more significant. To assure the dependability of WebAssembly runtime, the characteristics of WebAssembly runtime bugs are required to analyze in depth. Therefore, we conduct the first empirical study of WebAssembly runtime bugs by analyzing 867 bugs arising in four popular WebAssembly runtimes (V8, SpiderMonkey, Wasmer, and Wasmtime). We analyze the WebAssembly runtime bug characteristics based on their root causes, symptoms, bug-fixing time, and the number of files and lines of code involved in the bug fixes. From this study, we summarize 18 findings and provide an extended discussion of our main findings, which can guide future WebAssembly runtime bug detection, bug debugging, bug location, and bug repair.

Acknowledgment

We are very grateful to the anonymous reviewers for their thoughtful comments, which enable us to make improvements to the paper. This work is supported in part by the National Natural Science Foundation of China under Grants 62132020, 62072068, 62032004, 62202078, and Fundamental Research Funds for the Central Universities NO.NJ2020022.

References

[1] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1714–1730.

[2] M. Jacobsson and J. Willén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*. Springer, 2018, pp. 381–389.

[3] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[4] "V8," https://v8.dev/, 2017.

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

[6] "Awesome webassembly runtimes," https://github.com/appcypher/awesome-wasm-runtimes, 2018.

[7] E. Wen and G. Weber, "Wasmachine: Bring the edge up to speed with a webassembly os," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 353–360.

[8] S. S. Salim, A. Nisbet, and M. Luján, "Trufflewasm: a webassembly interpreter on graalvm," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 88–100.

[9] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: Webassembly as a practical path to library sandboxing," *arXiv preprint arXiv:1912.02285*, 2019.

[10] A. Prokopec, "Announcing graalwasm — a webassembly engine in graalvm," https://medium.com/graalvm/announcing-graalwasm-a-webassembly-engine-in-graalvm-25cd0400a7f2, 2019.

[11] S. S. Salim, A. Nisbet, and M. Luján, "Towards a webassembly standalone runtime on graalvm," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2019, pp. 15–16.

[12] P. Ventuzelo, "Fuzz testing in webassembly vms," https://medium.com/wasmer/fuzz-testing-in-webassembly-vms-3a301f982e5a, 2020.

[13] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. Chan, "Wasmfuzzer: A fuzzer for webassembly virtual machines," 2022.

[14] P. Ventuzelo, "A journey into fuzzing webassembly virtual machines," https://fuzzinglabs.com/journey-fuzzing-webassembly-wasm-vm/, 2022.

[15] "Spidermonkey," https://firefox-source-docs.mozilla.org/js/index.html, 2017.

[16] "Wasmer," https://wasmer.io/, 2018.

[17] "Wasmtime," https://Wasmtime.dev/, 2017.

[18] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[19] "Wasi." https://wasi.dev/, 2019.

[20] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui, "An exploratory study of autopilot software bugs in unmanned aerial vehicles," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 20–31.

[21] S. Chaliasos, T. Sotiropoulos, G.-P. Drosos, C. Mitropoulos, D. Mitropoulos, and D. Spinellis, "Well-typed programs can go wrong: a study of typing-related bugs in jvm compilers," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[22] 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*, 2016, pp. 294–305.

[23] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021.

[24] "Github api v3." https://developer.github.com/v3/, 2019.

[25] "Linking a pull request to an issue." https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue.

[26] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark and taxonomy of javascript bugs," *Software Testing, Verification And Reliability*, vol. 31, no. 4, p. e1751, 2021.

[27] J. Chen, Y. Liang, Q. Shen, and J. Jiang, "Toward understanding deep learning framework bugs," *arXiv preprint arXiv:2203.04026*, 2022.

[28] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 968–980.

[29] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, Chen, and Q. Alfred, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 385–396.

[30] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[31] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi, "Analyzing bug fix for automatic bug cause classification," *Journal of Systems and Software*, vol. 163, p. 110538, 2020.

[32] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 149–157.

[33] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[34] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 2020, pp. 604–620.

[35] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[36] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable? A study of api misuse on stack overflow," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 886–896.

[37] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 464–467.

[38] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu, "An empirical study on api-misuse bugs in open-source c programs," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 11–20.

[39] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[40] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 267–278.

[41] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Software Engineering*, vol. 23, no. 5, pp. 2901–2947, 2018.

[42] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[43] J. H. Zar, "Spearman rank correlation," *Encyclopedia of biostatistics*, vol. 7, 2005.

[44] M. Medeiros, U. Kulesza, R. Bonifacio, E. Adachi, and R. Coelho, "Improving bug localization by mining crash reports: An industrial study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 766–775.

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

[46] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 348–357.

[47] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.

[48] 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, 2014.

[49] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[50] "Wasm3," https://github.com/wasm3/wasm3, 2019.

[51] "Wasmedge," https://github.com/WasmEdge/WasmEdge, 2019.

[52] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.