

An Empirical Study of Bugs in WebAssembly Compilers

Alan Romano

University at Buffalo, SUNY
alanroma@buffalo.edu

Xinyue Liu

University at Buffalo, SUNY
xliu234@buffalo.edu

Yonghui Kwon

University of Virginia
yongkwon@virginia.edu

Weihsang Wang

University at Buffalo, SUNY
weihsangw@buffalo.edu

Abstract—WebAssembly is the newest programming language for the Web. It defines a portable bytecode format for use as a compilation target for programs developed in high-level languages such as C, C++, and Rust. As a result, WebAssembly binaries are generally created by WebAssembly compilers rather than being written manually. To port native code to the Web, WebAssembly compilers need to address the differences between the source and target languages and dissimilarities in their execution environments. A deep understanding of the bugs in WebAssembly compilers can help compiler developers determine where to focus development and testing efforts. In this paper, we conduct two empirical studies to understand the characteristics of the bugs found in WebAssembly compilers. First, we perform a qualitative analysis of bugs in Emscripten, the most widely-used WebAssembly compiler. We investigate 146 bug reports in Emscripten related to the unique challenges WebAssembly compilers encounter compared with traditional compilers. Second, we provide a quantitative analysis of 1,054 bugs in three open-source WebAssembly compilers, AssemblyScript, Emscripten, and Rust/Wasm-Bindgen. We analyze these bugs along three dimensions: lifecycle, impact, and sizes of bug-inducing inputs and bug fixes. These studies deepen our understanding of WebAssembly compiler bugs. We hope that the findings of our study will shed light on opportunities to design practical tools for testing and debugging WebAssembly compilers.

I. INTRODUCTION

WebAssembly is the newest language for the Web. Since appearing in 2017 [1], many prominent tech companies and news websites, such as eBay, Google, Norton, and CNN, have adopted the technology for various use cases such as barcode reading [2], video players, and TensorFlow.js machine learning applications [3]. Currently, WebAssembly is supported by major browsers including Chrome, Firefox, Safari, and Edge.

WebAssembly defines a portable bytecode format that serves as a compilation target for high-level languages such as C/C++ and Rust, enabling developers to port native applications to the Web. Rather than being written manually, WebAssembly bytecode is usually created by WebAssembly compilers such as Emscripten [4] or Rustc (with Wasm-Bindgen) [5].

As WebAssembly is increasingly adopted for various applications, there is a growing ecosystem of compilers that support WebAssembly development. As shown in Table I, there are currently 10 compilers available to support compiling programs written in different programming languages to WebAssembly [6].

Similar to compilers of native languages, WebAssembly compilers also contain bugs that can miscompile binary

TABLE I
STATISTICS OF COMPILER PROJECTS.

Compiler	Created	Source	LOC	Releases	Stars
AssemblyScript	2017-09	TypeScript	140,548	131	10,935
Emscripten	2011-02	C/C++	226,196	363	20,191
Rustc/Wasm-Bindgen	2010-06	Rust	1,013,824	98	58,276
Asterius	2017-11	Haskell	92,078	0	1,537
Binaryen	2015-10	asm.js	115,106	190	4,597
Bytecode	2017-04	Java	151,601	29	333
Faust	2016-11	Faust DSP	320,723	48	1,199
Ilwasm	2015-08	.NET CIL	4,549	0	344
Ppci-Mirror	2016-11	Python	12,998	0	174
TinyGo	2018-06	Go	4,910	20	7,399

outputs [7]. These bugs are difficult to locate as they may be encountered only at project's runtime. Compiler bugs can also waste development time when debugging an affected project before realizing that the bug is introduced due to miscompilation. For these reasons, it is important to understand how reliable compiler projects are in discovering, understanding, and resolving bugs.

In addition to handling the bugs associated with traditional compilers, the developers of WebAssembly compilers face unique challenges that can introduce buggy behavior. For example, fully synchronous executions are not natively supported by browser engines, which differs from the execution model expected by C/C++. WebAssembly compiler developers should ensure that synchronous operations in C/C++ code are properly ported over to the asynchronous browser environment as relying on asynchronous APIs to perform synchronous behavior can lead to issues. Moreover, JavaScript does not support all data types supported by WebAssembly. WebAssembly compilers have to support the compilation of data types across multiple target languages, as well as ensuring that, during runtime, types are not used in incorrect ways.

In this paper, we perform an empirical analysis of bugs in WebAssembly compilers to investigate the following research questions:

- RQ1: What new challenges exist in developing WebAssembly compilers and how many bugs do they introduce?
- RQ2: What are the root causes of these bugs?
- RQ3: How do WebAssembly compiler developers reproduce these bugs and what information is needed?
- RQ4: How do WebAssembly compiler developers fix bugs?
- RQ5: How long does it take to fix bugs in different compilers?
- RQ6: What are the impacts of the bugs in diverse compilers?

To answer these research questions, we first perform a

TABLE II
FINDINGS AND IMPLICATIONS OF OUR STUDY.

Findings	Implications
1 Data type incompatibility bugs account for 15.75% of the 146 bugs (Section IV-B2).	Interfaces (e.g., APIs) passing values between WebAssembly and JavaScript caused type incompatibility bugs when their data types are mishandled in one of the languages. Such interfaces (e.g., <code>ftell</code> , <code>fseek</code> , <code>atoll</code> , <code>labs</code> , and <code>printf</code>) require more attention.
2 Porting synchronous C/C++ paradigm to event-loop paradigm causes a unique challenge (Section IV-B1).	While automated tools support the synchronous to event-loop conversion (e.g., Asyncify), bugs in them may cause concurrency issues (e.g., race condition, out-of-order events). Programs going through this conversation require extensive testing.
3 Supporting (or emulating) linear memory management models is challenging (Section IV-B3).	WebAssembly emulates the linear memory model (of the native execution environment). Many bugs reported in this regard require a particular condition (e.g., allocation of a large memory to trigger heap memory size growth), calling for more comprehensive testing.
4 Changes of external infrastructures used in WebAssembly compilers lead to unexpected bugs (Section IV-B4).	Compiler developers should stay on top of developments that occur in the existing infrastructure used within the compiler. In particular, valid changes (in one context) of existing infrastructure can introduce unexpected bugs in WebAssembly. Rigorous testing is needed.
5 Despite WebAssembly being platform independent, platform differences cause bugs (Section IV-B8).	The default Emscripten Test Suite focuses on testing V8 browser and Node.js, while there are bugs reported due to the platform differences (e.g., caused by other browsers and OSes). The test suite should pay attention to cover broader aspects of the platform differences.
6 Unsupported primitives not properly documented lead to bugs being reported in the compiler (Section IV-D9).	WebAssembly compiler developers should pay attention to keeping the document consistent with the implementation (e.g., mentioning <code>sigsetjmp</code> and function type bitcasting are not supported).
7 Some bug reports failed to include critical information, leading to a prolonged time of debugging (Section IV-C).	We observe that the current bug reporting practice can be improved. In particular, an automated tool that collects critical information (e.g., inputs, compilation options, and runtime environments) would significantly help in the bug reproduction process.
8 Bugs that manifest during runtime made up a significant portion (43%) of the bugs inspected (Section V-B).	Many bugs in the compilers cause runtime bugs in the compiled programs, which are more difficult to detect and fix. To mitigate these bugs, compiler developers should be sure to test the emitted modules in the test suites more exhaustively.
9 77.1% of bug-inducing inputs were less than 20 line and developers manually reduce the size of inputs (Section V-D).	In many cases, bugs can be successfully reproduced by relatively small inputs that are less than 20 lines. Currently, developers often manually reduce large inputs. Automated bug-inducing input reduction (e.g., delta debugging) would be beneficial.

qualitative study on 146 bugs in Emscripten to identify unique development challenges, and understand the root causes, bug reproducing and bug fixing strategies of these bugs. Next, we perform a quantitative study on 1,054 bugs among three WebAssembly compilers, namely AssemblyScript [8], Emscripten, and Rust/Wasm-Bindgen. This study focuses on the lifecycle of the bugs, their impacts, and the sizes of the bug-inducing inputs and bug fixes. Based on the findings obtained from the two studies, we identify useful implications for WebAssembly compiler developers. Our findings and implications are summarized in Table II. We hope that our findings can provide WebAssembly compiler developers with specific areas that introduce bugs into the compiler, provide details on these bugs and previous fixes to help in designing new fixes, and provide general project management suggestions to prevent the introduction of new bugs.

II. WEBASSEMBLY DEVELOPMENT FLOW

WebAssembly defines an assembly-like bytecode format that is built to be fast and compact. The language also defines a text format meant to ease understanding. Specifically, the text format provides a readable representation of the module's internal structure, including type, memory, and function definitions. Unlike JavaScript, WebAssembly cannot access the Web APIs directly. Any reliance on these technologies such as the DOM, WebSockets API, and WebWorkers API requires complementary JavaScript code. At the minimum, WebAssembly requires JavaScript glue code to instantiate the WebAssembly module.

For our study, we define a WebAssembly compiler as a tool that can generate WebAssembly binary modules from source

code written in a high-level language. WebAssembly compilers are composed of a frontend that parses the source code into an intermediate representation (IR), an optional middle-end that optimizes the IR of the program, and a backend that generates WebAssembly binary code from the IR. In addition, WebAssembly compilers include bindings of existing libraries in order to support using standard libraries available in the source language within a WebAssembly runtime.

Fig. 1 shows a typical workflow of generating a WebAssembly program from *the source code in C++ to the runtime usage on a website*: (1) The C++ source program `example.cpp` shown in Fig. 1(a) defines a function `isEven()`. This C++ program is first compiled by a WebAssembly compiler, such as Emscripten (`emcc`), to generate the resulting WebAssembly binary `example.wasm` as shown in Fig. 1(b). The binary format is how a WebAssembly module is delivered to and compiled by browsers. (2) To ease debugging, the WebAssembly binary can be translated to its text format (`example.wat` shown in Fig. 1(c)) by using a WebAssembly toolkit, such as WebAssembly Binary Tool (WABT) [9]. The text format shows examples of WebAssembly instructions, such as `get_local` and `i32.and`, as well as the WebAssembly function `isEven()`. (3) To deploy the WebAssembly binary on a website, a JavaScript glue code as shown in Fig. 1(d) that instantiates the `example.wasm` file is required. The JavaScript code calls the function `WebAssembly.instantiateStreaming` that takes the parameter `fetch("example.wasm")` as the binary module source to instantiate. Finally, the returned module invokes the exported function `isEven()`.

WebAssembly modules are not typically standalone files. Instead, they are combined with generated JavaScript wrap-

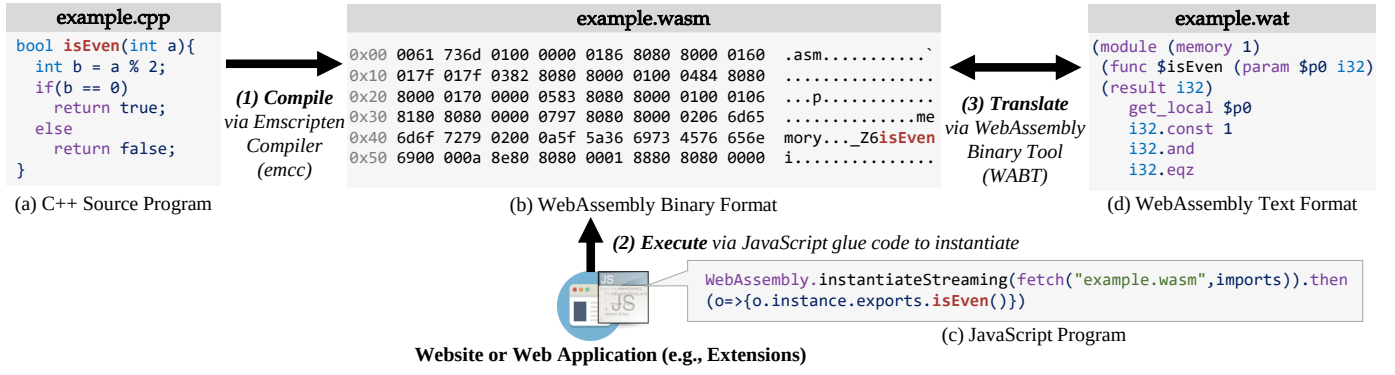


Fig. 1. WebAssembly Development Workflow.

per/glue code. Since WebAssembly cannot start on its own and cannot directly interact with WebAPIs, the glue code is responsible for importing the necessary functions used by the module. Additionally, the glue code can set up data structures necessary to implement the runtime provided by the native language, such as memory allocation, file system emulation, and socket emulation.

The final output of a WebAssembly compiler includes (1) a *WebAssembly module*, (2) a *JavaScript file* that handles the module imports and runs the module, and (3) an *HTML file* that loads the module.

III. DATA COLLECTION

A. Selecting WebAssembly Compilers

We inspect WebAssembly compiler projects on GitHub using the curated *awesome-wasm* list [6] that includes 10 WebAssembly compilers currently available, as shown in Table I.

We focus on popular compilers that support general-purpose, high-level programming languages. Specifically, we prune out *Faust* [10] (Domain Specific Audio DSP Language) and *Binaryen* (asm.js low-level target) as these source languages are not general or high-level. We also filter out compilers with less than 100,000 lines of code (*Asterius* [11], *Ilwasm* [12], *Ppci-Mirror* [13], and *TinyGo* [14]) and less than 50 releases (*Bytecoder* [15]) to focus on mature projects. To this end, our studies focus on three WebAssembly compilers, *Emscripten* [4], *Rustc/Wasm-Bindgen* [16], and *AssemblyScript* [8].

1. Emscripten compiles C/C++ to WebAssembly [17]. It originally targeted asm.js [18] – a precursor language to WebAssembly, so it precedes the creation of WebAssembly. It uses a modified Clang frontend and originally used Binaryen to provide the backend. It later adopted LLVM as the backend [19].
2. Rustc compiles Rust programs to WebAssembly [16]. As this compiler relies on the Wasm-Bindgen project [5] to provide bindings necessary for WebAssembly compilations, we include issues affecting both Rustc and Wasm-Bindgen in our count. We use the name Rustc/Wasm-Bindgen to highlight the combination of these two components.

3. AssemblyScript compiles a TypeScript-like language into WebAssembly [8]. It uses its own frontend and relies on Binaryen to handle the backend code generation.

In the qualitative study (Section IV), we aim to investigate WebAssembly compiler bugs in-depth to answer the research questions RQ1, RQ2, RQ3, and RQ4. For this purpose, we choose *Emscripten* because it is the most mature and widely-used WebAssembly compiler: (1) Emscripten was created earliest and has the most numbers of stars and milestone releases, compared with others. It also has the most number of reported bugs (will be discussed in Section V). (2) It dominates real-world usage [20].

B. Compiler Bug Collection

We collect bug reports from the three selected WebAssembly compiler projects' GitHub repositories through two methods. First, we use the GitHub Search API [21] to collect closed GitHub issues related to WebAssembly¹. Second, we use the GitHub REST API [22] to collect all the issues and pull requests for the projects. We also collect the commits referenced in the timeline of each issue in order to find which files the issues affected in the repositories. After obtaining the full set of issues for each project, we use the keywords “bug”, “defect”, “error”, and “fault” to identify the issues likely to be bugs.

TABLE III
BUG REPORT DATASET.

Compiler	Start	End	Bugs	Unique Bugs	Commits
AssemblyScript	2018-02	2020-12	136	107	174
Emscripten	2015-06	2020-12	711	430	1,460
Rustc/Wasm-Bindgen	2017-12	2020-12	207	158	245
Totals			1,054	695	1,879

Qualitative Study Dataset (Emscripten). We extract all 430 closed bugs from the Emscripten project. We read the bug reports of these issues to only include those that are related to the challenges unique to WebAssembly compilers. Specifically, we check the root causes of the 430 bugs to determine whether a typical compiler targeting a native platform (e.g., GCC targeting x86-64) would need to deal with a similar root cause. If not, we consider that they are unique challenges to WebAssembly

¹Issues with keywords like “bug”, “good first bug”, and “breaking changes” with “WebAssembly”, “wasm”, and “wat”.

and include them in our dataset. This brings the final number of bugs to 146. This scale is on par with similar work involving manual inspection [3], [23].

Quantitative Study Dataset. As shown in Table III, we obtain a total of 1,054 bug reports and 1,879 related commits from the three compilers' GitHub repositories. The second and third columns show the earliest and latest dates of the bugs considered for the dataset. The number of bugs from each compiler (i.e., after applying the filters) and the number of commits relating to the bugs are presented in the fourth and fifth columns, respectively. Note that we exclude bugs earlier than June 2015 from consideration as these precede the development of WebAssembly [24]. Also, there are multiple bug reports for one single bug because they readdress previous issues for various reasons (e.g., incomplete previous fix).

IV. STUDY I: QUALITATIVE STUDY OF EMSCRIPTEN ISSUES

In the first study, we manually inspect Emscripten issues that contain bug-inducing code inputs to identify development challenges, bug causes, reproducing difficulties, and fixing strategies.

Fig. 2 presents the architecture of Emscripten. It is built on top of existing compiler tools and infrastructures with Clang being used to implement the frontend. LLVM is used to provide middle-end optimizations. Binaryen and LLVM provide the backend functionality. Although the three stages resemble a traditional compilation pipeline for C/C++ compilers, developers of Emscripten (and any WebAssembly compiler in general) face unique challenges. Specifically, Emscripten provides implementations of the standard C and C++ that emulate the functionality available on native platforms (e.g., file systems and threading). These emulation libraries implement the semantics of legacy system calls by leveraging functions from JavaScript runtime components. For example, the FS library in Emscripten emulates traditional filesystem operations within the browser. Additionally, Emscripten provides libraries that allow C/C++ to call JavaScript functions at runtime. This is done to allow the C/C++ code to interact with the DOM and Web APIs, which are only accessible through JavaScript. It also includes several utilities supporting compilation or optimization of the input rather than parts of the source language or libraries. At the end of the compilation, a WebAssembly binary module is emitted along with the JavaScript support code to provide a full WebAssembly package.

A. RQ1: Development Challenges

WebAssembly compiler developers face a set of challenges that are unique to the new language. We develop categories for these challenges using an inductive coding approach [25] where we create categories based on the description of the underlying root cause. From this description, we determine whether this is a common compiler issue [7] or an issue unique to WebAssembly features. We iteratively add and refine categories to form district groups. As shown in Table IV, we generalize 9 unique WebAssembly compiler development challenges.

TABLE IV
BUGS RELATED TO DEVELOPMENT CHALLENGES.

	Development Challenge	Count
1	Asyncify Synchronous Code	12
2	Incompatible Data Types	23
3	Memory Model Differences	12
4	Bugs in Other Infrastructures	25
5	Emulating Native Environment	23
6	Supporting Web APIs	17
7	Cross-Language Optimizations	15
8	Runtime Implementation Discrepancy	17
9	Unsupported Primitives	2
	Total	146

Challenge 1: Asyncify Synchronous C/C++ Code. Most basic operations in C/C++ are executed in a synchronous and blocking manner. However, fully synchronous executions are not supported by browser engines. Execution in browsers follows an event loop that does not block execution to allow user interactions [26], which differs from the execution model expected by C/C++. In order to support compiling to this model, WebAssembly compilers need to provide additional tools to handle converting synchronous blocking code to fit the event-based asynchronous browser environment. However, we find that the implementations of these tools can be incorrect or inconsistent, causing various bugs. We observe that 12 issues were introduced by these tools.

Challenge 2: Incompatible Data Types. We find 23 issues that are caused by incompatibilities in the data types passed between the multiple languages involved in Emscripten compilation. This includes type incompatibilities during compilation between C and WebAssembly and type incompatibilities at runtime between WebAssembly and JavaScript.

Challenge 3: Memory Model Differences. WebAssembly has a different memory model than native environments. These differences can lead to issues when compiling to WebAssembly, and we find that 12 issues can be attributed to these differences.

Challenge 4: Other Infrastructures' Bugs. Emscripten is built on top of existing compiler infrastructures and tools. As a result, bugs can be reported in the Emscripten repository but may be found to be caused in the tool of another infrastructure. These existing infrastructures include frontend parsers, backend code generators, and WebAssembly VMs (e.g., such as V8).

Challenge 5: Emulating Native Environment. Emscripten provides libraries to seamlessly emulate native environment features that are not available on the web. These include filesystems, POSIX threads, and sockets.

Challenge 6: Supporting Web APIs. In addition to emulating native environment libraries, Emscripten also provides APIs to support calling WebAPIs from C/C++ code. These WebAPIs include WebGL, the Fullscreen API, and IndexedDB, and these interfaces are called by existing C/C++ libraries such as OpenGL and SDL or by using the Emscripten-provided WebAPI bindings.

Challenge 7: Cross-Language Optimizations. Since Emscripten emits both a WebAssembly binary module and the supporting JavaScript runtime code, optimizers used on either output component must be able to collect usage information

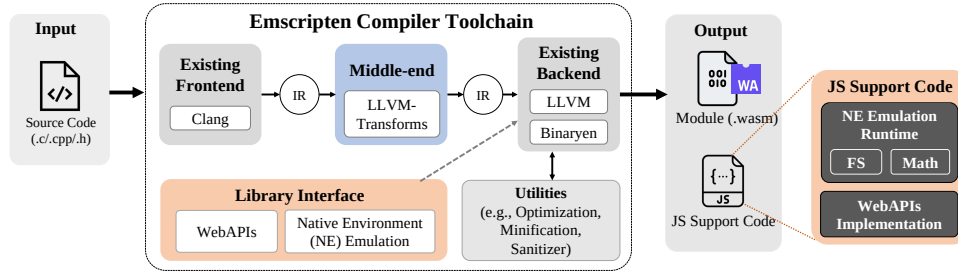


Fig. 2. Structure of Emscripten Compiler Toolchain.

TABLE V
ASYNCIFY SYNCHRONOUS C/C++ CODE BUGS.

Asyncify Tool	Causes	Count
Emterpreter	Emterpreter Parsing Errors	4
	Incorrect Emterpretify Stack State	2
	Missing Features in Emterpreter	3
Asyncify	Missing <code>sleep</code> Callback	1
Animation	<code>requestAnimationFrame</code> Misuse	1
IndexedDB	Flawed Filesystem Sync Operation	1
Total		12

from both languages. These optimizers can contain bugs that hinder the optimization of the resulting module.

Challenge 8: Runtime Implementation Discrepancy. Some issues can arise from differences in the running environment. This includes differences between browsers, differences in browsers and runtimes, and differences in runtimes supporting ES5 and/or ES6.

Challenge 9: Unsupported Primitives. Some issues arise when users attempt to perform functionality that touches on limitations in WebAssembly. For example, Emscripten does not support the C keyword `sigsetjmp`, because WebAssembly does not support signals [27].

B. RQ2: Bug Causes

We investigate the Emscripten bugs to identify and analyze the types of root causes among the issues. We read the conversation on the issue’s GitHub page to find what the developers reported the underlying issue to be. After identifying the root cause description for all issues, we generalize similar root causes into challenges listed in Table IV. We create root cause categories by using a deductive coding approach beginning with root cause categories from existing work [28]–[33]. We extend these categories to be more specific to WebAssembly compilers. To categorize these bugs, we read the issue reports to find what the compiler developers reported the underlying issues to be. We decide to which category the underlying root cause most relates to. Note that some root causes may be related to more than one category. For example, if the bug root cause is an invalid type operation from another infrastructure, we classify it as under *Incompatible Data Types*.

1) *Asyncify Synchronous Code Bug Causes*: There are 12 bugs in Emscripten tools that convert synchronous execution to asynchronous execution, as shown in Table V. Specifically, 4 bugs are caused by parsing errors in the Emterpreter tool. 2 bugs are caused by the internal state management of the

Emterpreter. 3 bugs are caused by unimplemented features in the Emterpreter. 1 bug is caused by the omitted `sleep` callback. 1 bug is caused by a misuse of the `requestAnimationFrame` browser function as a polling mechanism. 1 bug is caused by a flawed filesystem sync operation.

Fig. 3 gives an example [34] of the missing `sleep` callback bug. Emterpreter and Asyncify are two mechanisms provided by Emscripten to handle porting synchronous C/C++ code into event-based code compatible with the browser event-loop. Asyncify [19] allows for asynchronous execution by modifying WebAssembly code to allow for pausing and resuming during the middle of execution. Emterpreter [35] converts the input code into a bytecode format different from WebAssembly that is run in an interpreter that can be paused and resumed.

According to the documentation, both methods should perform *the same functionality*. This bug happens because the `emscripten_sleep` API in Asyncify behaves differently from the `emscripten_sleep_with_yield` function in Emterpreter. In particular, `emscripten_sleep` in Asyncify does not actually call a `sleep` callback. This difference leads to issues in the SDL library as it relies on these tools to handle streaming audio in the main loop. Audio chunks are enqueued through the `sleep` callback as shown in Fig. 3, so this lack of consistency leads to audio distortion in Asyncify.

```

1  SDL_AudioSpec as;
2  as.callback = audio_callback;
3  void audio_callback(void *unused, Uint8 *stream, int len) {
4      // push audio stream data to stream variable
5  }
6  while (true) {
7      // calculate audio stream data
8      emscripten_sleep_with_yield(1); // emscripten_sleep(1); -
9      ↪ for asyncify

```

Fig. 3. Emscripten Issue #9823: Missing `sleep` Callback.

This issue is fixed by adding the changes shown in Fig. 4 to the Asyncify library to call `sleep` callbacks, making it consistent with the behavior in Emterpreter.

2) *Incompatible Data Type Bug Causes*: We find 23 bugs within Emscripten that are a result of incompatible data types passed between the various languages involved in the compilation. As shown in Table VI, incompatible data type bugs result from root causes that can be grouped into three broad categories. The first group includes root causes involving native WebAssembly data types (i.e., `i32`, `i64`, `f32`, and `f64`). The

```

1 + sleepCallbacks: [], // functions to call every time we sleep
2 ...
3 handleSleep: function(startAsync) {
4 + // Call all sleep callbacks now that the sleep-resume is all
  → done.
5 + Asyncify.sleepCallbacks.forEach(function(func) {
6 +   func();
7 + });
8 ...
9 }

```

Fig. 4. Bug Fix for Emscripten Issue #9823.

TABLE VI
INCOMPATIBLE DATA TYPES BUG CAUSES.

Data Type	Causes	Count
Native	Incorrect i64 Legalization	7
	Unsupported Floating-Point or Precision Loss	4
	Missing i32 Operation	1
Custom	Incorrect C++ Atomic Types	4
	Invalid SIMD Type Operations	4
	Error Code Type Change	1
Undefined	Undefined Cross-Language Type Function	2
Total		23

second group involves types that are not native to WebAssembly, including C++ atomic types designed for threads [36], Single Instruction, Multiple Data (SIMD) values, and error code constants. The last category, *Undefined Cross-Language Type Function*, involves missing utility functions that fetch type information of compiled C/C++ values.

Fig. 5 gives an example [37] of the incorrect i64 legalization bug. This bug occurs when using a file pointer provided by the `stdio` library and compiling the module with option `-s MAIN_MODULE=1`. When compiling to WebAssembly, the browser sandbox prevents accessing the host filesystem. To get around this limitation, Emscripten provides a filesystem library, `FS`, implemented in JavaScript that emulates most of the functionality provided by `libc` and `libcxx`. The files are either provided as a static asset to download or embedded within the JavaScript wrapper. When the code in Fig. 5 is compiled, the calls to perform file I/Os are handled within this `FS` library on the JavaScript side.

```

1 #include <stdio>
2 int main() {
3     FILE* file_ = std::fopen("input.txt", "rb");
4     if (file_) {
5         std::fseek(file_, 0L, SEEK_END);
6         std::fclose(file_);
7     }
8     return 0;
9 }

```

Fig. 5. Emscripten Issue #9562: Incorrect i64 Legalization.

Since JavaScript does not natively support 64-bit integers, passing the 64-bit integer values to JavaScript is usually handled by a method called *legalization* which converts the 64-bit value into two 32-bit integers holding low and high bits separately. Within the execution path to `fseek()`, an indirect call attempts to pass a WebAssembly i64 value to exported WebAssembly function of a side module. The issue is that this other module's export function has been wrapped in JavaScript code to support value legalization, so although the first module knows that

the export function's type is i64, the intermediate JavaScript function cannot accept the parameter.

The issue is fixed by exporting legalized and non-legalized versions of WebAssembly functions so that function calls made through the indirect calls used here can pass i64 values to the appropriate function when legalization is not required.

3) *Memory Model Difference Bug Causes*: We observe 12 bugs in Emscripten that are a result of the differences in memory model between WebAssembly and a native environment. In a native environment, memory is allocated directly from the main memory, while WebAssembly uses the data structures available in the host VM to allocate a block of memory to function as the module's linear memory.

Specifically, there are 5 bugs that do not update the memory location after the memory is relocated. 2 bugs are caused by unnecessarily disabling memory growth in combination with another functionality, such as building standalone modules. 1 bug does not free unused resources after they are no longer needed, resulting in increased memory usage. Another bug attempts to access memory beyond the intended range. 1 bug shifts the boundaries of a memory buffer incorrectly. A bug incorrectly leaves zero-filled memory regions in the initial memory file, increasing the size of the file.

Fig. 6 shows an example [38] of the missing reference update bug. When a WebAssembly program allocates a large amount of heap memory, the memory might be relocated to a different location. If a program stores a memory location and does not update the location after the heap memory is relocated, it will cause a runtime exception because it refers to an invalid memory location.

This bug occurs when both WebAssembly memory growth and file system functionality are used (e.g., via `MEMFS` filesystem). When the WebAssembly module is initialized, the file content is stored in the heap section created in the module memory. The `MEMFS` filesystem is one of the Emscripten-emulated filesystems, and it supports in-memory file storage. It contains a reference to this location for future file operations. After the `malloc(20000000)`, the memory is grown, and the heap is moved to a different location. However, the filesystem reference is not updated, and the file contents cannot be read.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 void main() {
4     int c;
5     malloc(20000000); // Enlarge memory
6     FILE *fp = fopen("test.c", "r");
7     while ((c = fgetc(fp)) != EOF)
8         putchar(c);
9 }

```

Fig. 6. Emscripten Issue #5179: Missing Reference Updates.

The issue is fixed by forcibly enabling the '`--no-heap-copy`' flag, which stores the file system in a separate array to allow it to grow freely without worrying about filesystem references. However, this slows operations involving the `mmap()` syscall.

4) *Other Infrastructures Bug Causes*: We find 25 bugs reported in Emscripten that are caused by the tool of another infrastructure. The root causes of bugs related to the *Other*

Infrastructure Bug Causes challenge can be grouped by the infrastructure where the bug is caused. Specifically, we find 12 bugs affecting the LLVM Wasm Backend and 1 bug affecting the LLVM C++ standard library. There are 5 bugs affecting Binaryen and 1 bug in Clang. We also observe 3 bugs located in Firefox, 2 bugs in V8, and 1 bug in Safari.

For example, a bug was introduced into Emscripten when an update in Clang changed a default behavior when compiling with the options `-g3` or `-g4` (Emscripten Issue #7883 [39]). Previously, Emscripten would use value names found in LLVM IR to create the variable names in `asm.js`. An update in Clang discards these value names when generating the IR in order to improve the performance. The Emscripten developers were not aware of this change in Clang.

To fix this issue, the Emscripten developers utilize a new Clang flag (`-fno-discard-value-names`), which disables the new behavior and emits the value names in the IR.

5) *Emulating Native Environment Bug Causes*: We find that 23 bugs are related to the *Emulating Native Environment* challenge. Among the 23 bugs, 11 are in the emulated filesystem library and are caused by issues such as implicit dependencies, incorrect path resolving, and data truncation. 9 bugs are related to the pthread library and include issues such as thread scoping issues and incorrect termination. There are 3 bugs related to the socket library caused by issues such as unsupported functions.

6) *Supporting Web APIs Bug Causes*: We find that 17 bugs are caused by the challenge of *Supporting Web APIs*. 11 of the bugs are related to the WebGL APIs behavior not matching with OpenGL behavior. 2 bugs involving callback ordering are related to the Fullscreen API. 2 bugs impact the IndexedDB APIs by not handling possible errors. 2 bugs affect the WebAPIs exposed through the SDL library.

7) *Cross-Language Optimizations Bug Causes*: We find that the *Cross-Language Optimization* challenge produces 15 bugs. 9 bugs are caused by errors that lead to the optimizer marking a symbol as unused and removing it when it is needed by the code in the other language. 2 bugs are caused by syntactical mistakes in the optimizer code. 2 bugs are caused by errors in the optimizers variable scope tracking that prevent them from identifying all unused variables within the scope.

8) *Runtime Implementation Discrepancy Bug Causes*: Our results show that the *Runtime Implementation Discrepancy* challenge is responsible for 17 bugs. 5 of these bugs are related to Chrome API changes and behavior discrepancies. 1 bug is related to Safari and its immutable native objects. 1 bug is related to unsupported features in Internet Explorer. 2 bugs are related to NodeJS are caused by misusing V8 or built-in module APIs. 4 bugs are related to runtimes that do and do not support ES6 are caused by the behavior changes made between ES5 and ES6, including module export immutability and new APIs being introduced. 4 bugs are related to other runtimes through causes such as lack of fallback support, API changes, and performance issues.

C. RQ3: Bug Reproducing Analysis

Reproducing a bug is often the first step of the debugging process. However, some bugs may require a particular input, environment setting, or compiler version. We analyze each bug report and conversations in the bug issue to understand the challenges in reproducing bugs. Moreover, we inspect bug reports whether they contain all the critical information or not.




Information in Bug Reports. Table VII presents the critical information for bug reproduction included or discussed in the issues we check. “ID” represents the GitHub issue IDs. “Src.” indicates the source code of a program that causes the bug. “JS” means the JavaScript code snippet required to run the compiled WebAssembly program. “Stack” means a stack trace of the buggy program. “GT” represents the bug’s ground truth which includes the exact error message or expected values that can determine whether the bug is reproduced or not. “Opt.” and “Ver.” represent WebAssembly compiler options and versions used. “Env.” means the runtime environment (e.g., browser name and version). “Wasm” means the compiled WebAssembly program. The  symbol means that the information is provided in the bug report. Otherwise, they are not included in the report.

Table VII shows the results from a subset of the bug reports. A complete list can be found on [40]. Observe that most bug reports include sources, stack traces, ground truth, and compiler options, while information for compiler versions and runtime environments are relatively less frequently included. Moreover, compiled WebAssembly files are rarely included. Our manual investigation shows that those reports including WebAssembly files are typically high-quality reports. From those observations, we realize that an automated approach to create informative bug reports is highly desirable. Specifically, when a compiler generates a WebAssembly program, information for all the columns of Table VII can be collected to create a bug report file, similar to memory dump files containing various information about the environment.

Bug Reports Lacking Information. We further investigate bug reports that required significantly more effort in reproducing the bugs. Table VIII shows such cases. Note that we introduce the  symbol to represent information added after the initial report, requested mainly by the developers. It takes some time for developers to request additional information for the reports with many  symbols.

Compared with Table VII, sources (“Src.”) and ground truth (“GT”) are not frequently included in those reports, while those are critical in reproducing the bugs. Compile options and compiler versions are not well provided, and none of the reports includes compiled WebAssembly programs. The last five cases are the ones that developers express difficulty in reproducing. In particular, the initial report of #7409 lacks critical information, leading to many conversations with the developer to provide the missing information.

Overall, our analysis shows that many of the bug reports lack sufficient information to quickly reproduce reported bugs. We find that this lack of information can lead to longer debugging time in the compiler project. Compiler developers should

explore methods to ensure as much useful information is reported in these bug reports.

TABLE VII
INFORMATION INCLUDED IN THE BUG REPORTS.

Cat.	ID	Src. ¹	JS ²	Stack ³	GT ⁴	Opt. ⁵	Ver. ⁶	Env. ⁷	Wasm ⁸
Incompatible Data Types	3487	☑	☑	☑	☐	☑	☑	☐	☐
	3787	☑	☐	☑	☑	☑	☐	☐	☐
	3788	☑	☐	☑	☑	☑	☐	☐	☐
	3789	☑	☐	☐	☑	☐	☐	☐	☐
	3849	☑	☑	☐	☑	☐	☐	☐	☐
	3892	☑	☐	☐	☑	☐	☑	☑	☐
	4251	☑	☐	☐	☑	☑	☑	☑	☐
	5031	☐	☐	☐	☐	☐	☐	☐	☐
	5370	☑	☐	☐	☐	☑	☑	☐	☑
	6309	☑	☐	☑	☐	☐	☐	☐	☐
	7199	☑	☑	☐	☑	☑	☑	☐	☑
	7208	☐	☐	☑	☑	☑	☑	☐	☐
Asyncify Synchronous Code	3141	☑	☐	☑	☑	☑	☐	☐	☐
	3908	☑	☐	☐	☑	☐	☑	☐	☐
	4046	☐	☑	☑	☐	☑	☐	☑	☐
	5716	☑	☐	☑	☑	☑	☐	☐	☐
	6724	☑	☐	☑	☑	☑	☑	☐	☐
	6727	☑	☐	☑	☑	☑	☐	☐	☐
	6738	☑	☐	☑	☑	☑	☐	☐	☐
	6804	☐	☑	☐	☐	☐	☐	☐	☐
	6818	☐	☐	☑	☐	☐	☐	☑	☐
	7988	☐	☐	☐	☐	☐	☐	☐	☐
	9823	☑	☐	☐	☑	☑	☑	☐	☐
	10051	☑	☐	☐	☑	☐	☐	☑	☐
Memory Model Differences	3636	☑	☐	☑	☑	☑	☑	☑	☐
	3907	☑	☐	☐	☑	☑	☑	☑	☐
	5179	☑	☑	☑	☑	☑	☑	☑	☐
	5187	☑	☑	☐	☑	☑	☑	☑	☐
	5585	☐	☐	☐	☐	☑	☐	☐	☐
	6359	☑	☐	☐	☑	☐	☐	☐	☐
	7409	☑	☐	☑	☑	☑	☑	☑	☐
	8637	☑	☑	☐	☑	☑	☑	☐	☐
	9497	☐	☑	☐	☐	☐	☐	☐	☐
	9587	☑	☑	☑	☐	☑	☑	☐	☐
	9808	☑	☐	☐	☐	☐	☐	☐	☐
	10179	☑	☐	☑	☑	☐	☐	☐	☐

1. ☑ means the source code is available.
2. ☑ the JavaScript support code is available.
3. ☑ means a stack trace is provided.
4. ☑ means the ground truth of the expected output is listed.
5. ☑ means the compiler options used are listed.
6. ☑ means the version of the compiler used is listed.
7. ☑ means the runtime information is provided.
8. ☑ means the compiled WebAssembly binary is available.

D. RQ4: Bug Fixing Strategies

We investigate the different strategies used to fix these Emscripten bugs. We determine the bug fix strategy by reading the issue conversation to see if the developers explicitly mention the fix used. If it is not mentioned, we inspect the fixing commit, which is the last commit before the issue is closed. For each compiler challenge category, we group bugs with similar fixes into categories developed using an inductive coding approach on the fix descriptions. Note in all categories we omit low-frequency categories.

1) *Asyncify Synchronous Code Bug Fix*: The fixing strategies used to resolve the *Asyncify Synchronous C/C++ Code* issues can be grouped by the tool used. Bugs caused by a fault in the Emterpreter tool were fixed by extending the internal state checking, improving the Emterpreter parsing, improving the

TABLE VIII
BUG REPORTS WHERE THE BUGS ARE DIFFICULT TO REPRODUCE.^{1,2}

ID	Src.	JS	Stack	GT	Opt.	Ver.	Env.	Wasm
3778	☐	☐	☐	☐	☐	☐	☑	☐
3857	☐	☑	☐	☑	☐	☐	☐	☐
3861	☐	☐	☑	☐	☐	☐	☑	☐
3892	☑	☐	☐	☑	☐	☑	☑	☐
4046	☐	☑	☑	☐	☑	☑	☑	☐
4122	☐	☐	☑	☐	☐	☐	☐	☐
4646	☐	☑	☐	☑	☐	☑	☐	☐
5797	☐	☐	☑	☐	☐	☐	☑	☐
6169	☐	☑	☑	☐	☑	☐	☑	☐
6442	☐	☐	☑	☑	☑	☑	☐	☐
7146	☐	☐	☐	☑	☐	☐	☐	☐
7472	☐	☐	☑	☐	☑	☑	☑	☐
9091	☐	☑	☑	☐	☑	☑	☑	☐
9319	☐	☐	☑	☐	☐	☐	☐	☐
9650	☐	☑	☑	☑	☐	☐	☑	☐
10205	☐	☐	☑	☑	☑	☑	☐	☐
10317	☐	☐	☐	☐	☐	☑	☐	☐
10385	☐	☑	☑	☐	☑	☑	☐	☐
10675	☐	☑	☑	☐	☑	☑	☐	☐
3824	☑	☐	☐	☐	☐	☐	☐	☐
6534	☑	☐	☐	☐	☑	☐	☐	☐
7409	☐	☐	☐	☑	☐	☐	☐	☐
8001	☑	☑	☑	☑	☑	☑	☑	☐
10233	☐	☐	☑	☐	☐	☐	☑	☐

1. Refer to Table VII for column headers.
2. ☐ means the respective information was added after the initial post.

TABLE IX
ASYNCIFY SYNCHRONOUS C/C++ CODE BUG FIXES.

Asyncify Tool	Strategy	Count
Emterpreter	Improve State Checking	2
	Improve Emterpreter Parsing	4
	Improve Function Whitelisting	2
	Throw Exception	1
Asyncify	Add Sleep Callback	1
Animation	Add Warning Message	1
IndexedDB	Add Documentation	1
Total		12

function whitelisting functionality, or throwing an exception message to prevent misuse. Bugs caused by faults in the Asyncify tool were fixed by handling the missing `sleep` callback that led to inconsistent behavior compared with Emterpreter. Bugs caused by misuse of animation APIs were fixed adding a warning message against the incorrect usage. Bugs caused by a fault related to IndexedDB were resolved by updating the documentation to mention the buggy behaviors.

TABLE X
INCOMPATIBLE DATA TYPES BUG FIXES.

Data Type	Strategy	Count
Native	Fix/Bypass Legalization	5
	Add/Improve Type Support	4
	Add Documentation	2
	Provide Workaround	1
Custom	Fix/Remove Emitted Type Operations	6
	Change Type Used	2
	Provide Workaround	1
Undefined	Add Missing Function	2
Total		23

2) *Incompatible Data Type Bug Fix*: The bug fixing strategies applied on *Incompatible Data Types* bugs can be broken down by whether the root causes affected native WebAssembly types or special types not native to WebAssembly.

To fix *Native Types* issues, the following fixing strategies were applied. The *Fix/Bypass Legalization* strategy changes the JS-Wasm interfaces to either fix missing value legalization wrappers or disable unnecessary value legalization. The *Add/Improve Type Support* fixing strategy adds code to handle the unimplemented data types or improves the already-present code to handle a missing operation. The *Add Documentation* category describes the faulty behavior in the compiler documentation. In the *Provide Workaround* strategy, the compiler developers give the reporter a temporary solution to avoid triggering the fault while performing the originally intended action.

To fix issues involving *Custom Types*, the following strategies were applied. The *Fix/Remove Emitted Type Operations* fixing strategy changes the faulty code to either fix or remove the invalid type or related operations that caused the fault. *Change Type Used* fixes entail changing the variable type used that caused the faulty behavior.

The issues caused by *Undefined Cross-Language Type Functions* were fixed by adding the missing functions (*Add Missing Function* strategy).

3) *Memory Model Difference Bug Fix*: The bug fixing strategies applied to the 12 *Memory Model* bugs are as follows. 4 bugs fix the code obtaining memory references to change when the linear memory undergoes a change, such as memory growth. 2 bugs are fixed by releasing unused memory objects. 2 bugs change the operations calculating the boundaries of memory regions to prevent going out of them. 2 bugs change the allocation method used to remove the faulty method.

4) *Other Infrastructure Bug Fix*: The bug fixing strategies applied to *Other Infrastructure* bugs can be grouped by the related project. We find that the bugs in this challenge delegate the other infrastructure to fix the issue, including 11 wasm-ld bugs, 1 LLVM WebAssembly Codegen bug, 1 libcxx project bug, 2 V8 bugs, 3 Firefox project bugs, 3 asm2wasm bugs, and 1 wasm-opt bug. The *Use Workaround* fixing strategy is used on 3 bugs related to Binaryen, Safari, and Clang to avoid calling the code triggering the bug in the other infrastructure.

5) *Emulating Native Environment Bug Fix*: The bug fixing strategies applied to the 23 *Emulating Native Environment* are as follows. 6 bugs change the compiler options to automatically export the necessary dependency APIs when compiling. 3 bugs improve the functions gathering properties on the files or paths in the filesystem. 2 bugs remove any functionality that implicitly leads to a filesystem library dependency. 3 bugs change the code that sets global variables to also set those variables within the worker thread's scope. 2 bugs improve the release of used resources more reliably.

6) *Supporting Web APIs Bug Fix*: The fixing strategies applied to the 17 *Supporting Web API* bugs are as follows. 3 bugs add function cases to the list of supported WebGL extensions. 3 bugs wrap the faulty code in type checking to prevent accessing non-existent fields. 2 bugs were not fully

fixed by linked commits. 3 bugs change the event listeners used to avoid faulty behavior. 2 bugs impacting IndexedDB allow errors to be handled with try-catch statements rather than being hidden.

7) *Cross-Language Optimizations Bug Fix*: The fixing strategies applied to the 15 *Cross-Language Optimization* bugs are the following. 7 bugs change the code to prevent optimizers from changing the variable or field name so that it matches in both JavaScript and WebAssembly. 2 bugs add function definitions provided by the runtime environment to prevent the optimizer from marking the functions as undefined. 2 bugs emit a warning message describing the faulty behavior when it is called. 2 bugs correct typographical mistakes in the implementation of the optimizers.

8) *Runtime Implementation Discrepancy Bug Fix*: The fixing strategies applied to the 17 *Runtime Implementation Discrepancy* bugs are as follows. 5 bugs patch the affected code to avoid the runtime behavior discrepancies. 4 bugs add code to handle the cases where a feature is not implemented so that execution can continue. 3 bugs change the code logic to conform to updated runtime APIs. 3 bugs fix the checks that determine what properties or operations the environment has or supports. 2 bugs change the code to enable the use of particular runtime behaviors that improve performance.

9) *Unsupported Primitives Bug Fix*: The only fixing strategy applied to *WebAssembly Limitation* bugs is the *Provide Workaround* strategy to implement the unsupported functionality through different WebAssembly features.

V. STUDY II: QUANTITATIVE STUDY

In the second study, we perform a quantitative analysis on 1,054 bug reports collected from three compilers, AssemblyScript, Emscripten, and Rustc/Wasm-Bindgen, to understand the lifecycle of these bugs, the impacts that they have on compiled programs, the sizes of bug-inducing inputs, and the sizes of the fixes applied. The bug lifecycle shows how responsive compilers are in dealing with new bugs. Ideally, most bugs should be solved within one day [7]; however, our results show this is not the case. Investigating the impacts that bugs can cause on miscompiled programs is important in understanding the severity of the bugs that these compilers face. Understanding the sizes of bug-inducing inputs reveals the average code complexity needed to trigger bugs in these compilers, providing guidance for designing test cases. Inspecting the sizes of bug fixes reveals how widespread the bug impact is in the code.

A. RQ5: Lifecycle of Bugs

We analyze the duration between the time a bug is reported and the time the bug is fixed. We consider a bug as fixed when it is closed after a commit is referenced. If the bug is reopened, we use the time of the last closing event as the end of the duration. Fig. 7 presents the cumulative distribution of bug lifecycles. Rustc/Wasm-Bindgen, AssemblyScript, and Emscripten were able to fix 35.1%, 27.5%, and 23.6% of their bugs within 1 day, respectively. Within 10 days, the three compilers fixed over 50% of their bugs. These results show that the three compiler

projects fall short of the ideal same-day fix turnaround time [7]. This should be taken into consideration when deciding to use WebAssembly in a production-level project.

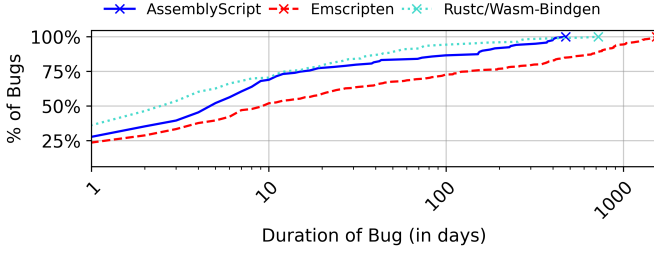


Fig. 7. Cumulative Distribution of Lifecycle of Bugs.

TABLE XI
IMPACT CATEGORIES FOR EACH COMPILER.

	Impact	Assembly Script	Emscr- ipten	Wasm- Bindgen
Build-Time	Build Error	23	54	18
	Compile Error	50	151	63
Compile-Time	Linker Error	0	2	22
	Code Bloating	0	12	1
Runtime	Crash	1	61	14
	Data Corruption	0	7	2
	Fail to Instantiate	2	5	4
	Performance Drop	1	2	3
	Hang	0	2	0
	Incorrect Functionality	5	59	14
	Other Runtime Error	25	75	17
	Total	107	430	158

B. RQ6: Impact of Bugs

We manually inspect all 695 unique bugs in the three compilers to find out whether the errors occurred at the compiler build time, program compile time, or runtime:

(1) *Build-Time Errors* prevent the compiler itself from successfully compiling [41], [42].

(2) *Compile-Time Errors* occur during the process that compiles source programs to WebAssembly binaries, including: (a) *Compile Error* fails to compile correct source programs (with no syntax errors) to WebAssembly binaries. (b) *Linker Error* fails to link the WebAssembly output with necessary libraries such as `stdlib`. (c) *Code Bloating* increases the size of the compiled WebAssembly file but does not affect the functionality [43], [44].

(3) *Runtime Errors* occur during the execution of a generated WebAssembly binary. The impacts of runtime errors include: (a) *Crash* leads to unrecoverable exceptions at runtime [45], halting the execution [46]. (b) *Data Corruption* corrupts the data stored in the output modules by losing or changing stored information [47]. (c) *Failure to Instantiate* fails to instantiate the WebAssembly binary because of inconsistencies with the wrapper code. (d) *Performance Drop* causes a noticeable slowdown in runtime performance when executing WebAssembly [48], [49]. (e) *Hang* stops responding to browser

events [50]. (f) *Incorrect Functionality* results in functionality inconsistent with what the compiled source code specified [51]. (g) *Other Runtime Error* does not fit into the above categories, such as missing debugging information [52]. Table XI shows the number of bugs by their impacts for the three compilers. We observe a significant portion of runtime errors. Specifically, 49.1%, 34.2%, and 31.8% of the bugs in Emscripten, Wasm-Bindgen, and AssemblyScript, respectively.

C. Bugs in Existing Compiler Projects

WebAssembly compilers often rely on components of existing compiler projects such as LLVM and Clang. We find 43 bugs are located in external compiler infrastructures, including 32 Emscripten bugs (16 LLVM, 12 Binaryen, 4 Clang), 10 Rustc/Wasm-Bindgen bugs (LLVM), and 1 AssemblyScript bug (Binaryen). Those bugs happen because the compiler developers misunderstand external projects' behaviors [53], [54] or updates on the external projects break assumptions made by developers [55], [56]. Note that the counts for Emscripten differ than those in Section IV due to differences in the bug selection criteria between the datasets.

D. Testing and Fixing Bugs

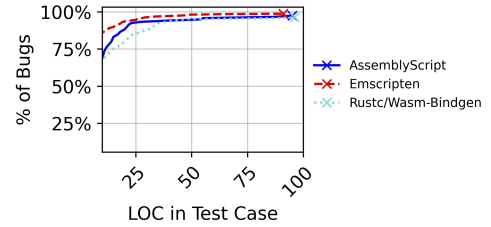


Fig. 8. Cumulative Distribution of Input Sizes.

1) *Size of Bug-Inducing Test Inputs*: Fig. 8 shows the distribution of the lines of code of the bug-inducing inputs that are given in the issue postings to reproduce the bugs, including source code and compiler options. Note that in our dataset, only 340 (48.9%) issues include the bug-inducing inputs. A large portion of bug-inducing inputs in all three compilers (183, 53.8%) have 10 or fewer lines of code, and 262 (77.1%) bugs-inducing inputs have 20 lines or fewer. In some cases, we observe a large program was provided initially as a bug-inducing input [57]–[59]. Later on, multiple posts on the same issue gradually developed to minimize the size of the bug inputs [60]–[62]. This suggests that techniques that can minimize testing inputs [63]–[66] are desirable.

2) *Size of Bug Fixes*: We analyze the size of bug fixes in terms of the lines of code. Among all compilers, 58.4% of all bugs (Emscripten: 43.7%, Rustc/Wasm-Bindgen: 34.2%, AssemblyScript: 24.3%) of the bugs have been fixed with 10 or fewer LOC. Over 96% (Emscripten: 74.2%, AssemblyScript: 71%, Rustc/Wasm-Bindgen: 69%) of all bug fixes have 100 or less LOC. On the other hand, two compilers, AssemblyScript (213.6 LOC) and Emscripten (189 LOC), have the bug fixes with an average LOC greater than 100. These large fixes are usually the result of the compiler developers incorporating

many changes into a single commit, rather than relating to the complexity of the issue. For example, in AssemblyScript, a bug involving missing functions when importing from a file was fixed in the same commit the developer cleaned the project, inflating the lines of code changed [67].

VI. DISCUSSION

Our findings can be found in Table II, and we highlight the most insightful ones here. We also discuss the threats to validity and a limitation in our bug fix identification strategy. **Qualitative Study's Findings.** Through our qualitative study, we find several interesting trends in the Emscripten compiler bugs. Finding 1 shows that *Incompatible Data Types* bugs make up 15.75% of the 146 bugs inspected. We find many of these bugs originate from interfaces relating to string handling (e.g., `printf`) and filesystems (e.g., `fseek`) rather than numeric interfaces. This finding can help developers diagnose similar bugs that arise by providing them with code locations to investigate. Finding 4 shows that changes and bugs in existing infrastructures can cause bugs. Compiler developers need to follow the development of leveraged infrastructures more closely to prevent these bugs. Finding 7 reveals that many bug reports fail to include critical debugging information, including the compiler version, environment, or source code used that triggers the bug. Compiler developers should include automatic reporting tools to include this information when submitting a bug report to aid in debugging.

Quantitative Study's Findings. Through our quantitative study, we obtain some insights into these compiler projects. For example, Finding 9 shows that 77.1% of bug-inducing inputs used were less than 20 lines of code, and developers frequently reduce this manually. This suggests that many bugs in these compilers can be reproduced by small inputs, which favors the use of automated input reduction techniques.

Threats to Validity. Similar to other empirical studies, our study is potentially subject to several threats, namely the representativeness of the chosen compilers, the generalization of the studied bugs, and the correctness of the analysis methodology. Regarding the representativeness of the chosen compilers, we choose three compilers that are the most popular and actively maintained WebAssembly compiler projects.

Another threat concerns the generalization of the studied bugs. We uniformly use all bug issues satisfying the selection criteria stated in Section III-B. We exclude bugs that were found to be irrelevant to WebAssembly after manual inspection. To ensure correct results, we only study fixed bugs because unfixed or unconfirmed reports may not be real bugs.

Regarding the correctness of the analysis methodology, aside from the analysis of test case LOC and impact, we automate all other analyses mentioned in this paper. The manual inspections on bugs to identify the sizes of test cases and impacts might be biased due to our inference of the test cases. To reduce this threat, three authors analyzed these bugs separately and discussed inconsistent results until an agreement was reached. **Sizes of Bug Fixes.** Bug fixes may also contain feature updates that are not relevant to the bugs. Moreover, fixes for some

design bugs require significant changes in the underlying code base, resulting in large bug fixes. In general, identifying bug-fix relevant parts from a software patch is a challenging problem. In our paper, we do not aim to distinguish this, and we observe a few such cases result in large bug fixes. However, from our manual inspection results shows that those are exceptional cases, and they do not affect our key findings and observations.

VII. RELATED WORK

Empirical Studies on Software Defects. Much research effort has been made to study fault related characteristics of software systems [7], [68]–[76]. For example, Sun et al. [7] conducted the first empirical study on the characteristics of the bugs in two mainstream compilers, GCC and LLVM. Tan et al. [30] inspect bug root causes, impacts and components to find characteristics within open-source projects. Eyolfson et al. [77] study open-source project commits to identify correlations between bugginess and commit-time characteristics.

WebAssembly Analysis Tools. There are a few tools made to analyze WebAssembly security and execution. Wasabi [78] is a framework to perform dynamic analysis on WebAssembly code by instrumenting the binary files to insert analysis code. Szanto et al. [79] and Fu et al. [80] perform taint tracking on WebAssembly to identify possible input vulnerabilities in a module. WasmView [81] visualizes the interaction between WebAssembly and JavaScript for a web application. WATT [82] is an authoring tool that helps create WebAssembly libraries. WASim uses machine-learning to automatically identify the purpose of a WebAssembly module [83].

WebAssembly Prevalence and Security Studies. Prior work [84] conducts a study on the prevalence of WebAssembly in the Alexa Top 1 Million websites. Hilbig et al. [20] study over 8,000 samples from various sources, including websites, GitHub repositories, and package managers. Lehmann et al. [85] analyze the binary security of WebAssembly and find weaknesses introduced by WebAssembly compilers. WebAssembly compiler frameworks and changes to the specification have been proposed to protect against Spectre [86], enhance memory safety [87], and support constant-time operations [88]. Finally, previous security works have focused on detecting in-browser cryptominers [89]–[91].

VIII. CONCLUSION

We conduct two empirical studies. In the first study, we perform a qualitative analysis on 146 bugs in Emscripten and analyze their root causes. We conduct a quantitative analysis on 1,054 bugs in three open-source WebAssembly compilers and reveal various aspects of these bugs. Our code and data set are publicly available at <https://wasm-compiler-bugs.github.io/>.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was partially supported by NSF under awards 2047980, 1916499, 1908021, 1850392, and a Mozilla Research Award (2019). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

- [1] 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*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 185–200. [Online]. Available: <http://doi.acm.org/10.1145/3062341.3062363>
- [2] S. Padmanabhan and P. Jha, "Webassembly at ebay: A real-world use case," 2020. [Online]. Available: <https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/>
- [3] A. Y. Daniel Smilov, Nikhil Thorat, "Introducing the webassembly backend for tensorflow.js," 2020. [Online]. Available: <https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html>
- [4] E. Contributors., "Emscripten," 2020. [Online]. Available: <https://github.com/emscripten-core/emscripten>
- [5] T. Rust and W. W. Group, "wasm-bindgen," 2020. [Online]. Available: <https://github.com/rustwasm/wasm-bindgen>
- [6] M. Basso, "mbasso/awesome-wasm," 2019. [Online]. Available: <https://github.com/mbasso/awesome-wasm>
- [7] 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*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 294–305. [Online]. Available: <https://doi.org/10.1145/2931037.2931074>
- [8] "Assemblyscript," 2020. [Online]. Available: <https://assemblyscript.org/>
- [9] W. C. Group, "webassembly/wabt," 2019. [Online]. Available: <https://github.com/WebAssembly/wabt>
- [10] "grame-cncm/faust," Jan. 2021, original-date: 2016-11-05T13:09:11Z. [Online]. Available: <https://github.com/grame-cncm/faust>
- [11] "Contents - asterius documentation," 2020. [Online]. Available: <https://asterius.netlify.app/>
- [12] K. Gadd, "kg/ilwasm," Jan. 2021, original-date: 2015-09-01T00:13:51Z. [Online]. Available: <https://github.com/kg/ilwasm>
- [13] tstreiff, "tstreiff/ppci-mirror," Jun. 2020, original-date: 2020-03-21T17:37:58Z. [Online]. Available: <https://github.com/tstreiff/ppci-mirror>
- [14] "Compiler internals," 2020. [Online]. Available: <https://tinygo.org/compiler-internals/>
- [15] M. Sertic, "mirkosertic/Bytecoder," Jan. 2021, original-date: 2017-04-13T10:21:59Z. [Online]. Available: <https://github.com/mirkosertic/Bytecoder>
- [16] T. R. Foundation, "The rust programming language," 2021. [Online]. Available: <https://github.com/rust-lang/rust>
- [17] E. Contributors., "Emscripten 1.39.4 documentation," 2020. [Online]. Available: <https://emscripten.org/>
- [18] A. Z. David Herman, Luke Wagner, "asm.js working draft," 2014. [Online]. Available: <http://asmjs.org/spec/latest/>
- [19] A. Zakai, "Emscripten and the llvm webassembly backend - v8," 2019. [Online]. Available: <https://v8.dev/blog/emscripten-llvm-wasm>
- [20] 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*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2696–2708. [Online]. Available: <https://doi.org/10.1145/3442381.3450138>
- [21] "Search - GitHub Docs," 2020, <https://docs.github.com/en/rest/reference/search>.
- [22] G. Inc., "GitHub api v3," 2019. [Online]. Available: <https://developer.github.com/v3/>
- [23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *Sigplan Notices - SIGPLAN*, vol. 47, 08 2012.
- [24] W. C. Group, "Roadmap," 2019. [Online]. Available: <https://webassembly.org/roadmap/>
- [25] X. Huang, H. Zhang, X. Zhou, M. A. Babar, and S. Yang, "Synthesizing qualitative research in software engineering: A critical review," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1207–1218. [Online]. Available: <https://doi.org/10.1145/3180155.3180235>
- [26] M. Contributors, "Concurrency model and the event loop - JavaScript | MDN," 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>
- [27] A. Trosinenko, "Usage of sigsetjmp/siglongjmp leads to undefined symbol references · Issue #5204," [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/5204>
- [28] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," 2005.
- [29] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," 01 2006, pp. 25–33.
- [30] 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, Dec. 2014. [Online]. Available: <http://link.springer.com/10.1007/s10664-013-9258-8>
- [31] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 55–64.
- [32] 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. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121220302740>
- [33] 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)*, 2015, pp. 101–110.
- [34] A. Guryanov, "Asyncify behaving differently then emterpreter · Issue #9823," [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/9823>
- [35] "Emterpreter — Emscripten 1.39.11 documentation," [Online]. Available: <https://emscripten.org/docs/porting/emterpreter.html>
- [36] "std::atomic - cppreference.com," [Online]. Available: <https://en.cppreference.com/w/cpp/atomic/atomic>
- [37] "-s MAIN_module=1 + upstream + function pointer calls using i64 => TypeError: cannot pass i64 to or from JS · Issue #9562," 2021. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/9562>
- [38] kichikuou, "Cannot access packaged files after memory growth · Issue #5179," [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/5179>
- [39] M. Zhang, "1.38.24 does not generate asm.js with correct variable names with -g3/g4 · Issue #7883," [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/7883>
- [40] "An Empirical Study on WebAssembly Compiler Bugs — wasm-compiler-bugs.github.io," 2020, <https://wasm-compiler-bugs.github.io/>
- [41] "Can't build incoming 64bits · Issue 4105," 2020, <https://github.com/emscripten-core/emscripten/issues/4105>.
- [42] "binaryen port failing to compile on windows · issue 4821," 2020, <https://github.com/emscripten-core/emscripten/issues/4821>.
- [43] "Exclude zero-initialized values from .mem file · Issue #3907," 2020, <https://github.com/emscripten-core/emscripten/issues/3907>.
- [44] "-mbulk-memory generates a 500 byte zero segment embedded into .wasm · Issue #8899," 2020, <https://github.com/emscripten-core/emscripten/issues/8899>.
- [45] "Crash using Haskell 'read' function · Issue 60 · tweag/asterius," 2020, <https://github.com/tweag/asterius/issues/60>.
- [46] V. Tikhonov, "A specific pattern of pushes to array causes a runtime error in lib/rtlsf/insertblock," 2020. [Online]. Available: <https://github.com/AssemblyScript/assemblyscript/issues/1042>
- [47] S. Seghers, "u32 interpreted as i32 when passed to js," 2019. [Online]. Available: <https://github.com/rustwasm/wasm-bindgen/issues/1388>
- [48] "Filesystem accesses from pthreads are much slower than in the main thread. · Issue 3922," 2020, <https://github.com/emscripten-core/emscripten/issues/3922>.
- [49] "[Question] upstream compiled binaries are 30% to 50% slower than fastcomp ones · Issue 9817," 2020, <https://github.com/emscripten-core/emscripten/issues/9817>.
- [50] K. Hiiragi, "Use pthreads freezes firefox nightly 42.0a1," 2015. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/3636>
- [51] GraDKh, "Webassembly: wrong conversion from double to int64 when outlining_limit isn't 0," 2018. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/6352>
- [52] "-g4 and wasm2js should generate JS source map or be an error · Issue 8743," 2020, <https://github.com/emscripten-core/emscripten/issues/8743>.
- [53] "Safe_heap=1 breaks source map generation for wasm," 2018. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/6534>
- [54] E. Cheng, "Bin-crate build fail with new lld linker in windows," 2018. [Online]. Available: <https://github.com/rust-lang/rust/issues/48948>

- [55] M. Zhang, “1.38.24 does not generate asm.js with correct variable names with -g3/g4,” 2019. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/7883>
- [56] S. Clegg, “Error on invalid archive member,” 2018. [Online]. Available: <https://github.com/emscripten-core/emscripten/pull/6961>
- [57] V. Tikhonov, “Runtime error during gc,” 2020. [Online]. Available: <https://github.com/AssemblyScript/assemblyscript/issues/1038>
- [58] makc, “abort(“alignment fault”) at jsstacktrace,” 2016. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/4760>
- [59] B. Vibber, “Assertion failure in wasm-ld linking libvpx with pthreads,” 2019. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/9155>
- [60] —, “error “variable fs is undeclared” in 1.39.7 with closure compiler,” 2020. [Online]. Available: <https://github.com/emscripten-core/emscripten/issues/10385>
- [61] A. Weissflog, “Code-gen bug related to 32-bit floats (actually: rand()),” 2016. [Online]. Available: <https://github.com/WebAssembly/binaryen/issues/817>
- [62] Y. Delendik, “Dwarf information does not contain absolute file locations for generics,” 2018. [Online]. Available: <https://github.com/rust-lang/rust/issues/54408>
- [63] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, p. 183–200, Feb. 2002. [Online]. Available: <https://doi.org/10.1109/32.988498>
- [64] Q. Luo, “Automatic performance testing using input-sensitive profiling,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1139–1141. [Online]. Available: <https://doi.org/10.1145/2950290.2983975>
- [65] S. Herfert, J. Patra, and M. Pradel, “Automatically reducing tree-structured test inputs,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 861–871.
- [66] M. Hörschele and A. Zeller, “Mining input grammars from dynamic taints,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 720–725. [Online]. Available: <https://doi.org/10.1145/2970276.2970321>
- [67] W. I. B. I. found on WasmBoy, “Weird importing bug i found on wasmboy,” 2018. [Online]. Available: <https://github.com/AssemblyScript/assemblyscript/issues/29>
- [68] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” *Operating Systems Review (ACM)*, vol. 35, 09 2001.
- [69] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *ASPLOS*, 2008.
- [70] S. K. Sahoo, J. Criswell, and V. Adve, “An empirical study of reported bugs in server software with implications for automated bug diagnosis,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 485–494. [Online]. Available: <https://doi.org/10.1145/1806799.1806870>
- [71] F. Thung, S. Wang, D. Lo, and L. Jiang, “An empirical study of bugs in machine learning systems,” in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 271–280.
- [72] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, “How do fixes become bugs?” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 26–36. [Online]. Available: <https://doi.org/10.1145/2025113.2025121>
- [73] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, “An empirical study on configuration errors in commercial and open source systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 159–172. [Online]. Available: <https://doi.org/10.1145/2043556.2043572>
- [74] 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*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: <https://doi.org/10.1145/3213846.3213866>
- [75] J. Aranda and G. Venolia, “The secret life of bugs: Going past the errors and omissions in software repositories,” in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 298–308.
- [76] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang, “Characterization of linux kernel behavior under errors,” in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, 2003, pp. 459–468.
- [77] 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. [Online]. Available: <https://doi.org/10.1007/s10664-013-9245-0>
- [78] 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*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1045–1058. [Online]. Available: <https://doi.org/10.1145/3297858.3304068>
- [79] A. Szanto, T. Tamm, and A. Pagnoni, “Taint tracking for webassembly,” *CoRR*, vol. abs/1807.08349, 2018. [Online]. Available: <http://arxiv.org/abs/1807.08349>
- [80] W. Fu, R. Lin, and D. Inge, “Taintassembly: Taint-based information flow control tracking for webassembly,” *CoRR*, vol. abs/1802.01050, 2018. [Online]. Available: <http://arxiv.org/abs/1802.01050>
- [81] A. Romano and W. Wang, “Wasmview: Visual testing for webassembly applications,” in *Proceedings of the 42nd International Conference on Software Engineering Companion*, ser. ICSE’20 Companion. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3377812.3382155>
- [82] H. Jeong, J. Jeong, S. Park, and K. Kim, “Watt : A novel web-based toolkit to generate webassembly-based libraries and applications,” in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2018, pp. 1–2.
- [83] A. Romano and W. Wang, “Wasim: Understanding webassembly applications through classification,” in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1321–1325.
- [84] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, “New kid on the web: A study on the prevalence of webassembly in the wild,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 23–42.
- [85] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of webassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann>
- [86] S. Narayan, C. Disselkoe, 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 *USENIX Security Symposium*. USENIX, August 2021.
- [87] C. Disselkoe, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, “Position paper: Bringing memory safety to WebAssembly,” in *Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, June 2019.
- [88] J. Renner, S. Cauligi, and D. Stefan, “Constant-time WebAssembly,” in *Principles of Secure Compilation (PriSC)*, January 2018.
- [89] 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*, ser. CCS ’18. New York, NY, USA: ACM, 2018, pp. 1714–1730. [Online]. Available: <http://doi.acm.org/10.1145/3243734.3243858>
- [90] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, “Outguard: Detecting in-browser covert cryptocurrency mining in the wild,” in *The World Wide Web Conference*, ser. WWW ’19. New York, NY, USA: ACM, 2019, pp. 840–852. [Online]. Available: <http://doi.acm.org/10.1145/3308558.3313665>
- [91] A. Romano, Y. Zheng, and W. Wang, “Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1129–1140. [Online]. Available: <https://doi.org/10.1145/3324884.3416580>