C to WebAssembly Analysis

Anonymous Author(s)

ABSTRACT

WebAssembly is an increasingly popular, portable, low-level bytecode language designed to work with JavaScript in the browser. WebAssembly allows browsers to support CPU-intensive applications, and the Emscripten compiler makes it possible to cross-compile traditional C/C++ desktop applications to the web. Currently, transparent**[TS: This might be due to my lack of knowledge, but I'm not sure what transparent porting is, maybe it would be useful to add an explanation at some point.]** porting is still hard to accomplish - practitioners need to modify the application source code to make it fit the WebAssembly architecture. In this paper, we investigate what misbehavior can occur after porting C programs to WebAssembly. We collect 14,024 valid C programs from three sources, compile them into WebAssembly using Emscripten, and compare their execution outputs with those of the original C programs. 54 of these programs exhibit different execution outputs. We manually inspected these discrepant results and identified nine root causes. For each type of cause, we provide advice to developers on how to handle them in practice. In conclusion, porting existing C programs to WebAssembly for cross-platform distribution still needs code adaptations. More effort should be put into stitching inconsistencies between WebAssembly and other high-level languages.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**; **Assembly languages**.

KEYWORDS

WebAssembly, Differential Testing, Compiler

ACM Reference Format:
Anonymous Author(s). 2022. C to WebAssembly Analysis. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

WebAssembly is an increasingly popular portable low-level bytecode language that offers compact representation and safe low to no-overhead execution [22]. Implemented by all major browsers in 2017 [52], WebAssembly is supported by 95% of all global browser installations as of Sep 2022[1]. For a long time, JavaScript, as the only option for developing web applications, is criticized for its subpar performance, while the advent of WebAssembly gives browsers the opportunity to embrace CPU-intensive applications, such as games. As a low-level language, it has better loading speed and runtime performance. João *et al.* [15] conduct the first systematic study showing that WebAssembly improves energy efficiency by 30% on average compared with JavaScript.

WebAssembly is designed as a compilation target of other high-level languages. There are several widely used compilers, *e.g.,* Emscripten[2] for C and C++, or the Rust compiler, making it possible to port applications or libraries which are written in languages other than JavaScript to the browser. It has been seven years since W3C built the WebAssembly Community Group [48], however, WebAssembly is not widespread as we expected: only limited desktop applications have been successfully ported to the web, such as AutoCAD [49], Photoshop [50], and Acrobat [5]. One reason for this is that it's hard to ensure the execution behavior keeps correct when compiling to WebAssembly. **[WW: Any evidence this is the reason? e.g., citing some bug reports or articles]** Compared with native binary, WebAssembly has a different language design and runtime architecture, which impose unique challenges on WebAssembly compilers. For example, the infinite loop in games or multimedia applications, which is used to continuously update the graphic frame, will lead to a crash when compiled to WebAssembly and run in a browser. **[WW: So no infinite loops is used in WebAssembly game applications?]** Instead, developers need to rely on callbacks into the WebAssembly code from the web browser. **[WW: Until here, the work is not well-motivated. More evidence is needed to support why such a study is important. Is this a prevalent problem? What consequence such discrepancy leads to? How severe it is? What motivates you to study this problem? Is there any unique characteristic of WebAssembly makes such a study more interesting?]**

In this paper, we conduct research on potential discrepancies that can occur when compiling C code to WebAssembly code through cross-compiler differential testing. Specifically, we investigate the compilation from C to WebAssembly using Emscripten, the most commonly used WebAssembly compiler to date, and GCC. We ran tests to detect differences in the execution result of the output of these two compilers, and manually analyze the different ones to find out the root cause leading to the discrepancies.

A primary challenge for the experiment is to collect a large quantity of high-quality C programs for testing. **[WW: I think this shouldn't be considered as a challenge.]** Given that Emscripten does not have full support for C features such as file manipulation and multi-threading, we chose to use small, easily-to-compile single-file programs as test subjects. These files' main purpose should be to write to the standard output and not rely on anything outside the C standard libraries. Based on this requirement, we synthesize

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1]https://caniuse.com/?search=WebAssembly
[2]https://emscripten.org/

previous articles regarding C compiler testing and propose three approaches to collect test programs: mutation-based C test suite, log-inserted GCC test suite, and auto-generated C programs using Csmith [3]. In the last, we get 14,024 C test files in total and detect 54 files existing execution output inconsistency between WebAssembly and native binary. Through manual inspection, we divide them into six categories based on the root causes: long type size, static memory allocation, stack arrangement, read-only data protection, unsupported language feature, and infinite loop. For each category, we describe based on case studies and present code details, then we give the advice to help developers correctly handle these differences in practice. In summary, this paper makes the following contributions:

- We generate a dataset of 14,024 valid single-file C programs suitable to compiler testing, which is public online.
- We identify six root causes for difference in execution of C programs compiled to WebAssembly and to native binary.
- We discuss several examples to illustrate behavior differences' cause, and give advice to help developer correctly handle these differences in practice.

The remainder of this paper is organised as follows. Sec. 2 provides necessary background to understand WebAssembly language and the motivation of this paper. In Sec. 3, we discuss our approach to collect C programs and conduct differential testing. The findings of the experiment are listed in Sec. 4. Sec. 5 discusses all the threats to the validity of our experiment. Related work are put in Sec. 6. Finally we presents our conclusion in Sec. 7.

2 BACKGROUND AND MOTIVATION

2.1 Differences between C binary and WebAssembly

[WW: Based on the subsection title, this subsection should talk about the differences between C and WebAssembly. However, it only describes execution model and memory of WebAssembly, but not that of C. Even adding C descriptions, more explanation is needed as for why these differences matter to this paper.]

Execution Model The execution model of WebAssembly is stack-based: instructions push values on and pop values from a value stack. Values can be stored in local variables and global variables, which are similar to registers. Arguments for function calls are passed through the stack; *i.e.,* they need to be pushed on the stack beforehand. The return value of a function is the top value of the stack after executing the instructions in its body. Apart from the value stack which we described, there is no need to manually manage the call stack in WebAssembly for function calls. The actual jumping to and returning from function bodies is entirely managed by the runtime.

Memory Architecture A WebAssembly application contains a single linear memory, *i.e.,* a consecutive sequence of bytes that can be read from and written to by specific instructions. Using this linear memory properly is left to the program at hand. Hence, the runtime does not restrict in any way the usage of this linear memory. For example, there is no concept of page or segment in the linear memory, unless these are implemented in the program being executed directly.

[3]https://embed.cs.utah.edu/csmith/

2.2 Difficulty in Porting to WebAssembly

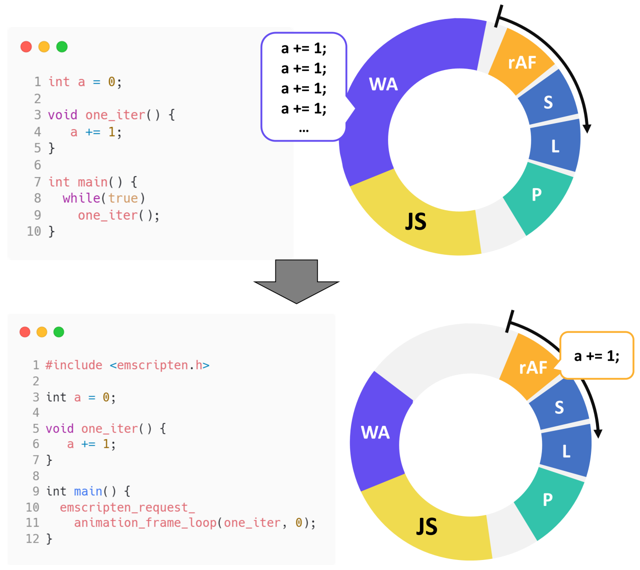
[WW: The flow of this subsection is unclear. What are the challenges when compiling C to WebAssembly? Can you categorize? Also, please separate examples from categories.]

WebAssembly, along with C toolchain compilers (such as Emscripten), has provided programmers with a way to port traditional C programs to the Web. However this process is tricky, especially when it comes to large C applications. There exists several factors which make the porting process difficult for developers, and can result in the changing of programs' behavior.

The first category is due to the fact that some C libraries does not have a WebAssembly version of the implementation, or has a different implementation. For example, the multi-threading is not yet available in WebAssembly. So for the programs depending on thread libraries, such as Photoshop[50], porting to WebAssembly would lead to different runtime behavior and lower performance. Besides, some libraries has different implementation compared with native version, which may also lead to difference of runtime behavior.

Apart from issues caused due to library implementations, the WebAssembly language standard has inherent features that can cause programs behaviour to be altered. WebAssembly has a novel execution model, memory architecture and runtime requirements. One obvious difference between C and WebAssembly is the handling of infinite loop. C/C++ apps with graphical interfaces typically run in an infinite loop. Within each iteration of the loop the app performs event handling, processing and rendering, followed by a delay (wait) to keep the frame rate constant. However, this infinite loop is a problem in the browser environment. Fig. 1 illustrates how to the infinite loop should be adapted to suit the browser main loop (also called event loop), which is represented in the diagram as a coloured circle. In the loop, browser needs to first go through several rendering steps: "S" as style calculation, "L" as layout calculation, "P" as pixel painting. "rAF" is where the *requestAnimationFrame* function will be called by the browser. This method provides ability to perform an animation and requests that the browser calls a specified function to update an animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint. After these steps, then browser will start to handle JavaScript event (yellow segment in Fig. 1). In our example, the JavaScript event will instantiate WebAssembly module compiled from the C code (left side of the circle), and execute it immediately. In the upside code snippet, we put an infinite loop in the main function, and for each iteration "a += 1" will be executed. Under this situation, the WebAssembly module will repeatedly execute "a += 1" operation and never yield the control back to browser main loop, causing subsequent render steps being blocked. After a period of time the browser will notify the user that the page is stuck and offer to halt or close it.

Facing this problem, Emscripten provides an API *emscripten_request_animation_frame_loop* to get the environment to call this same function at the "rAF" stage (orange segment). The iteration is still



```
int a = 0;

void one_iter() {
    a += 1;
}

int main() {
  while(true)
    one_iter();
}
```

```
#include <emscripten.h>

int a = 0;

void one_iter() {
    a += 1;
}

int main() {
  emscripten_request_
    animation_frame_loop(one_iter, 0);
}
```

Figure 1: How Infinite Loop Fits in the Browser Main Loop. Two code snippets in the plot will be compiled to WebAssembly and run in the browser. For the upside situation, UI rendering jobs are blocked by the infinite loop in WebAssembly module. When modified to the downside situation, it changes the infinite loop using Emscripten provided API, thus each loop iteration is executed in "rAF" stage and page freezing is prevented.

run "infinitely" but now other code can run between iterations and the browser does not hang.

Library implementation deficiencies can be fixed gradually, while the misbehavior caused by language feature will not change in the short term and requires special attention from developers. In this paper, we further investigate what kind of misbehaviour will happen when porting to WebAssembly due to language inherent feature, and we focus on programs only depends on C standard libraries that are fully implemented by Emscripten.

3 APPROACH

3.1 Overview

[WW: Can you add steps in the text and figure?] [WW: What is preprocessing?] Our approach's workflow is shown in Fig. 2. In our experiment, we collect program dataset from three different sources: C test-suite, GCC test-suite, and Csmith. For the first two test-suite, in order to increase the code diversity, we use mutation method to enlarge the test-suite size. We combine all the test files together as the "C Test Program Dataset". For each C program in the dataset, we compile it using Emscripten and GCC respectively. Emscripten will generate WebAssembly file together with JavaScript glue code to help it run in browser environment. Then we run them on three mainstream browsers: Chrome, Firefox, and Edge. We run the output of GCC on native OS. In the last, we collect their runtime standard outputs, and examine whether difference exists. If difference are found, the C program will be flagged and manual check will be applied later to determine the root cause.

3.2 Dataset Collection

The quality and quantity of C programs decides whether we can find valuable inconsistency between C binary and WebAssembly. In order to collect large numbers of C programs as our experiment input, we find three dataset sources and use mutation method to increase the code diversity. The left half of Fig. 2 shows our dataset collection progress. Given that Emscripten does not have full support for C features such as file manipulation and multi-threading, we choose to use small, easily-to-compile single-file programs as test subjects. These files' main purpose should be to write to the standard output and not rely on anything outside the C standard libraries. Test suites coming from three sources are introduced as followings:

1. C test-suite "C test-suite"[10] is a collaborative database on GitHub of C compiler test cases, minimal test runners, and public test results. It is designed to test C compilers, consisting of 220 small test files, each focused on a single aspect of the C compilation process.

2. GCC test-suite GCC provides a collection of C codes to test compilers' functionality [47]. Since these tests don't rely on external libraries, GCC test-suite is frequently used in previous compiler testing works [43]. In our experiment, we use test files under the "gcc.dg" folder, 9,888 tests in total, of which we filtered out 1,251 tests. We got this sub set by filtering out all tests with a Main function, as this makes the compilation process to WebAssembly easier to implement. We then removed several tests from this sub-set if they were not able to compile, for example, 5 tests were removed for containing non utf-8 characters. This left us with a test suite size of 1,251. Further more this test suite is under the care of the GCC team, and targets fault points in the C compilation process, increasing the likelihood that our tests will find discrepancies in behaviour between C and WebAssembly.

3. Test-suite Generator There are many mature test program generator. Among them, Csmith is one widely used fuzzing tool which can generate valid C programs automatically. It iterates through C grammars and create statements randomly based on given seed. Csmith will calculate the checksum of all variables' value in the end and set it as the return value. So it's convenient to find out runtime inconsistency just by comparing the returned checksum. In our experiment, we keep generating new test code by passing new seed to Csmith, and filtering out code with lines less than 100. In this way, we collect 10,000 auto-generated valid C programs.

3.3 Mutation Method

Mutation generation is used in our approach to expand "C test-suite" and "GCC test-suite". This was done to increase the quantity and variability of tests. To enlarge the test-suite size, we used the MULL[16] mutation tool. MULL is an open source tool for mutation testing created by Alex Denisov and Stanislav Pankevich. We selected this mutation tool due to its ease of implementation, and its extensive documentation. MULL works by creating mutations of a program in memory using LLVM bit code, and then inserts these altered lines of code back into the original program, thus producing mutants. Mutants could have been generated on any basis, as they were used as a way to find areas where errors may

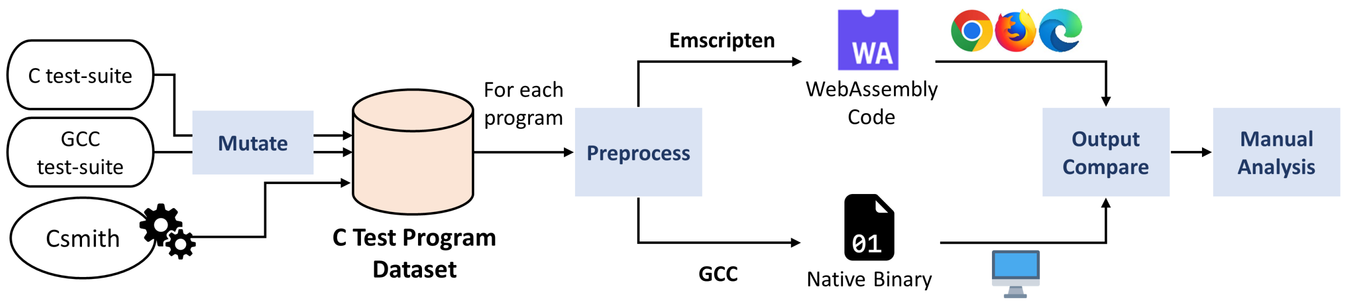


Figure 2: Workflow of Compiler. We collect C test programs from three sources. Each C test file is compiled using Emscripten and GCC respectively. And the execution outputs of the outputs of two compilers will be compared.

occur. This paper used the supported Mull operations[39] to mutate our test suite. This includes basic arithmetic substitutions(such as replacing '+' with '-'), logic substitutions (such as replacing '==' with '!='), inverting incrementals ($x++$ to $x--$), and constant substitutions ($a == b$ to $a == 42$). Such mutations can have unpredictable results on the behaviour of a program; ranging from unnoticeable, to causing the code to crash.

In our experiment, we expand "C test-suite" from 220 to 624 tests and "GCC test-suite" from 1,251 to 3,400 tests through mutation. We combine them with 10,000 tests generated from Csmith, finally we get a C program dataset consisting of 14,024 files in total.

3.4 Differential Testing

In our C program dataset, the main function's return value is the only program output. The return value is hard to retrieve when executing in browsers. Hence, in the preprocessing step of Fig. 2, we add data-collect hook for each program file. The hook consists of writing a script, which would insert a print statement above every return command. This allows us to automate how we collect output from the tests, as we could record the printed data. For C, this meant that after running the code in terminal, the output code could be collected and stored. For WebAssembly, we edited the html file to include Debugout.js. Debugout.js[38] was used to record the codes output to the web browser log. This output was then recorded as well. There were some problems with this approach. Such as if a pointer or memory address was returned, the test output would be different for C and WebAssembly. This would result in the test being incorrectly flagged, even though both programs were following expected behaviour.

After adding the hook, we compile each C program using Emscripten and GCC separately. We run the output of Emscripten on the latest version of three browsers: Chrome, Firefox and Edge. We run the output of GCC in native system terminal. We use SHELL script to automate the testing process. Script iterates through the C program files, takes them as input and compiles them using Emscritpen and GCC. For Emscripten, we run "emcc xxx.c -Ox -o test.wasm test.js", where the optimization level ranges from "-O0" to "-O4". Emscripten will generate a wasm file "test.wasm", together with a JavaScript glue code file "test.js" which helps invoking wasm functions in JS-based runtime. For GCC, we run it using default settings by "gcc xxx.c" and run the generated executable directly in terminal. Script will collect the execution outputs in terminal and compare using "diff" tool. If difference is detected, the C source code and Emscripten optimization level will be recorded for later analysis. We conduct our experiment on a machine with macOS Monterey system, and an AMD Ryzen Threadripper 3990X 64-Core CPU (2.9 GHz) with HyperThreading and 16 GiB of RAM. All the tools are using the latest stable version: Emscripten (3.1.23), Node.js (16.17.1), GCC (14.0.0).

4 FINDINGS

In this section, we present the results of our experiment, and categorise all the misbehavior we found into five types based on their root causes. These types are discussed from Sec. 4.2.1 to Sec. 4.3.4.

4.1 Experiment Results

In our experiment, we used data from three sources - "C test-suite", "GCC test-suite", and "Csmith"; with each containing a sample size of: 624, 3,400 and 10,000 C progams respectively. Among the total of 14,024 programs, 54 of them were flagged as behaving differently. With different behaviour being observed between Emscripten and GCC compile line. From manual inspection, we categorised these flagged programs into nine root causes, and divided them into following three categories based on the cause source:

1. Compiler Flagged programs under this category are caused by detail treatment difference between compilers. For example, although both C standard and WebAssembly Specification have descriptions about how variables should be arranged in stack during runtime, the accurate offset of variable in the stack fully depends on compilers' specific implementation. The location discrepancies of variables in memory may eventually lead to unexpected execution results. We found three causes in this category: "Stack Arrangement", "Function Pointer Cast", and "Builtin Function". In our experiment, 17 flagged programs belong to this one.

2. Runtime Flagged programs under this category are caused by difference in execution runtime - programs compiled by GCC run on a native machine, while programs compiled by Emscripten runs on browsers. The vast differences in runtimes mean that the code will inevitably need to be modified to fit the new environment. As we have discussed in Background (Sec. 2.2), infinite loops can work well when used in desktop applications, but they will block UI rendering steps when run in browsers and lead to page freezing. There are four causes under this category: "Memory Protection", "Multi-threading", “Large Loop”, and “Infinite Loop”. In our experiment, 30 flagged programs belong to this one.

3. Language Feature Flagged programs under this category are caused by difference in language features between C and WebAssembly. As a language still under development, many language features of WebAssembly are not yet fully implemented or are not yet supported by most runtimes. There are two causes under this category: “Long Type Size” and “Exception Handling”. In our experiment, 7 flagged programs belong to this one.

Table 1 is a breakdown of the flagged program number under each category. The following subsections in this chapter discuss the causes of each category in depth, illustrated with practical code examples and accompanied by advice for developers on the actual porting C to WebAssembly process.

4.2 Difference in Compiler

4.2.1 Stack Arrangement. When we visit out-of-bound index of an array, there is inconsistent runtime output between code generated by GCC and Emscripten. Listing. 1 is an example C code containing out of bound array visit.

```c
#include <stdio.h>

int main() {
    int array1[] = {10,20,30,40};
    int a = 1;
    int array2[] = {50,60,70,80};
    int b = 2;

    for (int i = -4; i < 8; i++) {
        printf("%d\n",array[i]);
    }
    return 0;
}
```

Listing 1: Array Overflow Example

In this program, two arrays: “array1” and “array2”, are defined and initialized with four elements separately. Two local variable “a” and “b” are defined as integer types.The program then iterates the index “i” from -4 to 7, and prints out the value of “array1[i]”. We compile this program with Emscripten and GCC, and their outputs can be seen in Table 2.

When index “i” is in bound, *i.e.,* in 0 to 4, the outputs from Emscripten and GCC keep same. While when “i” is out of bound, inconsistency happens. This is caused by different stack layouts generated by two compilers, which are shown in Fig. 3.

Fig. 3 shows the runtime stack frames of both runtime environments. In this figure, we assume that the address of stack frame base is 0. Notice that in Emscripten, the stack grows from the high address to the low address in default. Looking at the Emscripten stack frame, we can find that variables are arranged on the stack in the same order as they are declared in the source code. Because the “array2” needs 16 bytes alignment, it cannot be allocated immediately after the variable “a”. As such it is stored at -48 address. In contrast GCC reorders the variables in the stack frame and keeps them more contact and compact, and whilst maintaining memory alignment.The compilers allocate variables differently, and as such have different stack frame layouts. This leads to inconsistency of out-of-bound array visiting behaviour.

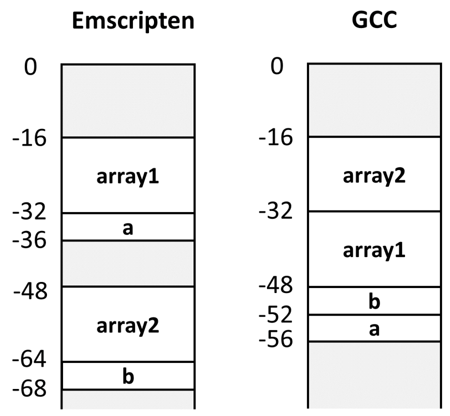


Figure 3: Stack Frames under Two Compilers. Although compiled from the same source code, the positions of variables are different.

4.2.2 Function Pointer Cast. It is undefined behavior in C to cast a function pointer to another type and call it that way. However, this does work in most native platforms, but fails in WebAssembly. Listing. 2 illustrates the function pointer cast issue. The code first defines a function *intReturn* (line 3) with “int” return type. This function is to print the message passed from the parameter. In the main function, the *intReturn* is converted to “void” return type (line 9) before being called. The program runs correctly when compiled by GCC. In contrast, when compiled to the WebAssembly, the changed function type will be considered as invalid and will abort when called in line 10.

```c
1  #include <stdio.h>
2
3  typedef void(*voidReturnType)(const char *);
4
5  int intReturn(char* msg) {
6    printf("%s",msg);
7    return 1;
8  }
9
10 int main() {
11   voidReturnType func = (voidReturnType)intReturn;
12   func("Hello World!");
13   return 0;
14 }
15
16 GCC: Hello World!
17 Emscripten: Invalid function pointer called.
```

Listing 2: Function Pointer Cast Example. Emscripten considers the function pointer cast as invalid.

4.2.3 Builtin Functions. Compiler builtin functions such as “__builtin_unreachable ()” behave differently due to the implementation discrepancy cross compilers.

4.3 Difference in Runtime

4.3.1 Memory Protection. One common runtime inconsistency we found in our experiment is due to the code trying to write read-only data in memory.

Table 1: Statistics of All Flagged Tests.

Due to Difference in ...	Root Cause	C test-suite	GCC test-suite	Csmith	Total
Compiler	Stack Arrangement (Sec. 4.2.1)	9	5	0	14
	Function Pointer Cast (Sec. 4.2.2)	0	1	0	1
	Builtin Function (Sec. 4.2.3)	0	2	0	2
Runtime	Memory Protection (Sec. 4.3.1)	6	2	0	8
	Multi-threading (Sec. 4.3.2)	0	1	0	1
	Large Loop (Sec. 4.3.3)	0	3	11	14
	Infinite Loop (Sec. 4.3.4)	3	2	2	7
Language Feature	Long Type Size (Sec. 4.4.1)	2	0	2	4
	Exception Handling (Sec. 4.4.2)	0	0	3	3
SUMMARY		20	16	18	54

Table 2: Array Overflow Example Output

Index i	Emscripten output	GCC output
-4	0	0
-3	0	0
-2	0	1
-1	1	2
0	10	10
1	20	20
2	30	30
3	40	40
4	0	50
5	0	60
6	0	70
7	0	80

Case1: String Literal Modification

In Listing. 3, we initialize a char pointer and assign "GfG" to it. Then we try to modify the second character of this string from 'f' to 'n'.

```
#include <stdio.h>

int main() {
  char *str = "GfG";  /* Stored in data segment */
  *(str+1)='n';
  printf("%s\n",str);
  return 0;
}

GCC: segmentation fault (core dumped)
Emscripten: GnG
```

Listing 3: String Literal Modification Example. WebAssembly doesn't provide memory protection to data segment.

This code will compile for both Emsripten and GCC. Yet, whilst the WebAssembly code will run successfully, the C code will run into a segmentation fault. This occurs, as when a string value is directly assigned to a pointer, it's stored in the data segment of the stack. This is a read-only block shared among functions. When one modifies read-only data in code compiled by GCC, the program will immediately terminate execution and report a segmentation fault. In contrast, this code runs fine when compiled with Emscipten, since there is no read-only restriction in WebAssembly's memory architecture.

Case2: Local Variable Return

The following code is similar to Listing. 3. It also uses a defined "str" variable initialized as "GfG". However, in this case, the string is declared as character arrays, so it is stored in stack segment. Then we return the address of this string, and print it out in main function. This results in differing behaviour.

```
#include <stdio.h>

char *getString() {
  char str[] = "GfG"; /* Stored in stack segment */
  return str;
}
int main() {
  printf("%s", getString());
  return 0;
}

GCC: segmentation fault (core dumped)
Emscripten: 0 p
```

Listing 4: Local Variable Return Example. WebAssembly doesn't provide memory protection to out-of-stack data.

If we run the code compiled by Emscripten, it will print some garbage data. This occurs as the data stored in the stack frame of function "getString()" may not be there after "getString()" returns. While the code compiled by GCC will directly throw out segmentation fault since the data outside stack top is also considered as read-only data, this does not occur in Emscipten.

Case3: Null Pointer Assignment

The following code is trying to assign a value to a null pointer.

```
#include <stdio.h>

int main() {
  int *junk = NULL;
  *junk = 567;
  printf("End\n");
  return 0;
}

GCC: segmentation fault (core dumped)
Emscripten: End
```

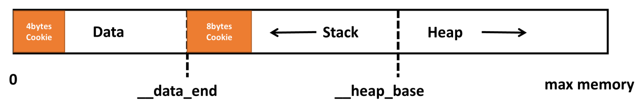


Figure 4: Security Cookies in WebAssembly Memory. Security cookies (orange) are added at the top of stack and data section.

```
12          Aborted (Runtime error: The application
        has corrupted its heap memory area(address
       zero)!)
```

Listing 5: Null Pointer Assignment Example. WebAssembly uses security cookie to detect stack overflow.

Assigning a value to a null pointer is equivalent to modifying the memory content at address 0. This is a read-only space under GCC memory architecture and will definitely leads to segmentation fault. As such, the "End" string in Listing. 5 will not be printed out. However, for Emscripten, it will "End" will get printed in console, but then the code will abort with runtime error.

Although there is no read-only restriction in WebAssembly's memory as we can see from case 1, Emscripten provides an additional protection mechanism for WebAssembly at runtime to prevent stack overflow - *security cookie*. Security cookie is similar to stack canary in GCC, which is a magic value added by the compiler to detect stack overflow. By default, Emscripten writes security cookies to the final two words in the stack and detects if they are ever overwritten. The global address 0 also sets a security cookie for integrity. These cookies will be checked before and after running the function. Fig. 4 shows the layout of WebAssembly memory, and the orange blocks are the cookies added by Emscripten.

Giving a value to a null pointer is the same as modifying the value of the cookie at address 0. So, in the Listing. 5 case, Emscripten does not terminate right after line 3, instead it checks security cookies after the function returns and then a runtime error is popped up.

A native executable will crash with a segmentation fault when facing an invalid memory operation. In contrast, WebAssembly doesn't provide similar memory language protection at language level. Emscripten adds memory protection logic in JavaScript glue code; security cookies will be added into linear memory before WebAssembly program starts. Unlike native C, which checks for invalid memory accesses in real time at the hardware level and terminates the program at the first sign of out-of-bounds, Emscripten only checks the values of cookies after WebAssembly main function returns. Other WebAssembly compilers such as Clang with wasm-ld backend doesn't provide any memory protection. The more lenient memory checking policy in WebAssembly leads to a number of behavior differences related to memory operations. Runtime stack overflow in WebAssembly is likely to be unnoticeable, which requires extra attention from developers.

4.3.2 *Multi-threading.*

Currently, threads is still an in-progress proposal and not supported by many WebAssembly runtime. As a result, directly invoking functions in "pthread" library will cause runtime error. Facing this shortage, Emscripten has implemented pthreads with shared memory. Although many limitations exists in its implementation, such as POSIX signals can not being supported, it allows multi-threaded C applications ported to the web. By default, support for pthreads is not enabled. Compiler flag *-pthread* is needed to enable code generation for pthreads. Besides, shared memory is disabled across browsers since January 2018 due to Spectre & Meltdown vulnerabilities, and needs to be manually enabled before using the pthreads.

```c
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *callback() {
5    /* Do Something*/
6  }
7
8  int main() {
9    pthread_t thread_id;
10
11   pthread_create(&thread_id, NULL, callback);
12   pthread_join(thread_id, NULL);
13   return 0;
14 }
15
16 GCC:
17 Emscripten: Tried to spawn a new thread, but the
       thread pool is exhausted.
```

Listing 6: Multi-threads Example. Thread can't be spawned due to the event loop limitation.

Listing. 6 is one simplified example of pthread usage leading to inconsistency in our experiment. In the main function, *pthread_create* (line 11) will create a background thread. It takes a destination to store a thread handle in, some thread creation attributes (here not passing any, so it's just NULL), the callback to be executed in the new thread, and an optional argument pointer to pass to that callback in case you want to share some data from the main thread — in this example we don't pass extra argument. *pthread_join* (line 12) is called later to wait for the thread to finish the execution, and get the result returned from the callback. It accepts the earlier assigned thread handle as well as a pointer to store the result. In this case, there aren't any results so the function takes a NULL as an argument.

We compile this code using emcc and pass a "-pthread" parameter. But when the *pthread_create* function is executed, a warning shows that "Tried to spawn a new thread" and the program aborts. What happened? The problem is, most of the time-consuming APIs on the web are asynchronous and rely on the event loop to execute. This limitation is an important distinction compared to traditional environments, where applications normally run I/O in synchronous, blocking manner.

In this case, the code synchronously invokes pthread_create to create a background thread, and follows up by another synchronous call to pthread_join that waits for the background thread to finish execution. However, Web Workers, that are used behind the scenes when this code is compiled with Emscripten, are asynchronous. So what happens is, pthread_create only schedules a new Worker thread to be created on the next event loop run, but then pthread_join immediately blocks the event loop to wait for that Worker, and by doing so prevents it from ever being created. It's a classic example of a deadlock.

One way to solve this problem is to create a pool of Workers ahead of time, before the program has even started. When pthread_create is invoked, it can take a ready-to-use Worker from the pool, run the provided callback on its background thread, and return the Worker back to the pool. All of this can be done synchronously, so there won't be any deadlocks as long as the pool is sufficiently large.

This is exactly what Emscripten allows with the -s PTHREAD_POOL_SIZE=... option. It allows to specify a number of threads—either a fixed number, or a JavaScript expression like navigator.hardwareConcurrency to create as many threads as there are cores on the CPU. The latter option is helpful when your code can scale to an arbitrary number of threads.

In the example above, there is only one thread being created, so instead of reserving all cores it's sufficient to use -s PTHREAD_POOL_SIZE=1.

4.3.3 Large Loop. Large loops widely exists among C programs, especially for computation-heavy applications. However, large loop in WebAssembly will greatly affect the smoothness and responsiveness of the web page. Smoothness requires a steady and sufficiently high frame rate, and Responsiveness requires the UI responds to user interactions with minimal delay. Since WebAssembly and JavaScript share the same execution thread, *i.e.,* the execution of WebAssembly will halt the JavaScript code, and vice-versa. According to RAIL[4], a user-centric web performance model introduced by Google, users perceive animations as smooth so long as 60 new frames are rendered every second, which means that browser only has 1000ms/60 = 16.6ms to produce each frame. It includes the time spent by browser to paint the new frame to the screen, leaving developers only about 10 ms to produce a frame. As a result, if the WebAssembly code can't finish job in 10ms and yield the control back to JavaScript, web users will feel a noticeable lag. And if the task on the WebAssembly side takes up to a few seconds or more, it will even freeze the current web page.

```
1  #include <stdio.h>
2
3  int main() {
4    int iterations = 1000000000;
5    float sum = 0;
6    for (int i = 0; i < iterations; i++) {
7      sum += i * 1.1 / (i + 1);
8    }
9    printf("%f", sum);
10   return 0;
11 }
```

Listing 7: Large Loop Example. Large loop will block browser UI rendering the freeze the web page.

Listing. 7 is a test program used in our experiment, which contains a large loop. This program takes 2 seconds to finish on native machine. When we compile it into WebAssembly and run in the browser, the web page crashes and can't get the output at line 7.

Facing the large loop issue, there are two common practices. The first one is to separate large loops to server side (back-end). This strategy requires extra effort to modify the C source code, *i.e.,* split large loops from original place and compile it separately to back-end code. Besides, additional JavaScript glue code is needed to transfer return value from back-end to front-end when loop is done. The second common practice is to wrap the WebAssembly code with *web workers.* JavaScript is designed to be single-thread, which causes that a piece of slow JavaScript code can prevent the browser's rendering loop from continuing. The advent of the web workers creates a multi-threaded environment for JavaScript, allowing the main thread to create worker threads and assign some tasks to the latter. Once the execution of WebAssembly is wrapped in a separate web worker, it will no longer affect the UI rendering job in JavaScript main thread. To date, support for web workers is already 98% [5]. However, their usage, according to a Google Dev Summit talk [46], is still relatively low. We recommend more web developers can combine WebAssembly with web workers to deliver computationally complex applications in the web environment. In May 2022, Adobe team presented how they port Photoshop, a complex image processing desktop application, to the web [44]. In order to exploit the full performance of WebAssembly, they made extensive use of web workers to make complex graphics processing algorithms run smoothly on the browser. Their experience validates the power of web workers on handling computation-heavy tasks.

4.3.4 Infinite Loop. Another type of large loop is infinite loop, a loop never ends. C++ apps with graphical interfaces typically run in an infinite loop. Within each iteration of the loop the app performs event handling, processing and rendering, followed by a delay (wait) to keep the frame rate constant. Infinite loop in WebAssembly will also crash the web page since the control will never return to the JavaScript.

For infinite loop, the standard solution is to define a C function that performs one iteration of the main loop without the "delay" (Many application will add "delay" in main loop in order to keep the frame rate constant). For a native build this function can be called in an infinite loop, leaving the behaviour effectively unchanged. For web build, Emscripten provides a function "emscripten_request_animation_frame_loop()" to get the environment to call this same function at the proper frequency for rendering a frame. The iteration is still run infinitely but now other code can run between iterations and the browser does not hang.

4.4 Language Feature

4.4.1 Long Type Size. Through our tests we found that WebAssembly does not allocate the same amount of memory to the "long" data type as C does. On a 64-bit operating system, the long type is assigned with 8 bytes under GCC, and 4 bytes under Emscripten. C will generally allocate a "long" 8 bytes, but will set a minimum of 4 bytes.

The reason for this difference is that Emscripten can currently only compile in 32-bit. Emscripten emits WebAssembly code using the upstream LLVM wasm backend, which only supports 32 bit wasm compile option. According to WebAssembly spec [42], "wasm64" option is a future planed expansion that would migrate WebAssembly to 64-bit. And "long" is 4 bytes size under 32-bit, 8 bytes size under 64 bit. As a result, there is difference in memory allocation for "long" data type between GCC and Emscripten.

4.4.2 Exception Handling. Exception catching is disabled in Emscripten by default. Listing. 5 is a simplified exception handling program, where the "throw" statement will abort the program under Emscripten. Exception catching functionality is necessary to

[4]https://web.dev/rail/

[5]https://caniuse.com/webworkers

maintain the normal flow of a program when undesired events occur. To overcome this limitation that exception syntax is not supported in WebAssembly, the Emscripten team makes an effort to handle the exception in JavaScript glue code, instead of in WebAssembly. However, this method significantly reduces the programs speed and is not recommended unless necessary. Developers can enable Emscripten's JavaScript-based exception support by passing "-fexceptions" option at both compile time and link time.

```
int main() {
  try {
      throw 1;
  } catch(...) {
      ...
  }
  return 0;
}

GCC:
Emscripten: exception thrown: 5246024 - Exception
     catching is disabled, this exception cannot be
      caught.
```

Listing 8: Exception Catch Example. When compiled using Emscripten, the "throw" (line 3) will abort the program.

4.4.3 Developer Advice. Up till now, many features like exception handling and threads are not a part of WebAssembly. Applications that throw exceptions need additional JavaScript code to fill the void. Utilizing these assist logic leads to increased code sizes and reduced performance. Fortunately, these in-progress features are gradually being filled at a faster rate than we thought - exception handling was officially released by Chrome in September 2021 [1]. Developers could refer to the WebAssembly Roadmap webpage [6], which lists all the finished proposals and ones in progress.

5 THREATS TO VALIDITY

5.1 Internal Validity

The first potential threat is bias in the selection of C test programs. Although we use three different methods to generate test programs to increase test-suite diversity, these tests cannot cover all possible code scenarios. So the root causes we identified may not be exhaustive. Most obviously, our test program does not exhaustively test the API provided by the C standard libraries. In our experiments, the test subjects only include a few common C standard libraries, such as *stdio.h*, *string.h*, and *pthread.h*. If more comprehensive test-suites are introduced, we believe more inconsistency could be found.

5.2 External Validity

One threat to internal validity lies in the selection of compiler. We performed our analysis using Emscripten to compile programs to WebAssembly. There are other compilers available, such as Clang with "wasm-ld" backend [7] and enterprise-grade compiler Cheerp [8]. Different compilers have different solutions for handling WebAssembly features. For example, Emscripten provides the implementation of multi-threading and exception handling through JavaScript glue code, while Clang doesn't. As a result, experimenting with compiler other than Emscripten may lead to completely different results.

Except the selection of compiler, WebAssembly runtime selection is also a threat. In our experiment, all the WebAssembly codes run in the latest version of three most widely-used browsers - Edge, Chrome and Firefox. Despite we didn't observe behaviour difference among these browsers, it does not exclude the possibility that WebAssembly will behave differently in other browsers, since every browser has the right to determine its WebAssembly support level and the details of the implementation strategy. The browser version will also influence the experiment result. As browsers gradually improve their support for WebAssembly, we believe that many of the existing porting issues can be better resolved in the future. Besides, browser is not the only runtime of WebAssembly. Although WebAssembly is designed to accelerate web application in the first place, the emergence of more and more runtimes allows WebAssembly to run on various platforms. One GitHub repository lists up to 33 known WebAssembly runtimes [9]. If WebAssembly runs on a platform other than a browser, it is not subject to the limitations of single-threaded rendering of JavaScript, and therefore there is no problem with large loops mentioned in Sec. 4.3.4.

Moreover, another threat to internal validity regards the source language. According to WebAssembly Developer's Guide [2], more than ten high-level languages can be compiled to WebAssembly. Each new language support brings unique challenges for WebAssembly compiler developers. It is also interesting to study what difficulty these languages face when porting to WebAssembly, which we leave for future work.

6 RELATED WORK

There are several areas of research which relate to our work:
Differential testing provides the same input to a set of equivalent applications, and can detect bugs in these systems by comparing each of the systems output [18, 32]. This technique has successfully be used to test space flight software [20] , test the Java virtual machine[13, 14], detect leaks of user information[26], determining runtime behaviour of code clones [54], locate errors in both the concrete and symbolic execution components of symbolic execution engines [27], and the verification of deep learning systems [21, 37].
Mutant Generation is the process of doing modifications to program X in order to generate X' a new program[11]. Mutant generation is most commonly used in mutation testing, a technique used to evaluate the quality of existing software tests[25, 29, 35]. The process of generating mutants has also been used for test generation. There are several types of mutation operation techniques, which include class operators and method operators[15]. Class operators, which is what we primarily used, mutate code by either by inserting, deleting or modifying the existing sections, usally based on some set of rules. Method operators mutate statements by inserting, replacing, or deleting primitive operators.
Test Generation is the process of generating test suites for a system[3, 4]. It has been used for testing mission software systems [51], control systems[30], web services [7], and even C compilers[53]. Yu-Wen Tung and Wafa S. Aldiwan use test generation for testing

[6]https://webassembly.org/roadmap/
[7]https://lld.llvm.org/WebAssembly.html
[8]https://leaningtech.com/cheerp/
[9]https://github.com/appcypher/awesome-wasm-runtimes

mission software systems [51], as it ensure large code coverage with a relatively small sample set of tests. J. Mattingley *et al.* proposes using test generation for control systems as allows for a custom solver. Our paper explores both test generation techniques, as well as mutant generation of existing test suites, to generate large sets of random input to our system. Mutant generation as a test generation technique has also been done before, for example, test generation for java applications [36], test generation for embedded software [17], and generation of unit tests [19].

WebAssembly Study WebAssembly has become increasingly popular in both research and industry in recent years. It has been used for cryptomining[8, 28, 33], games[8], software libraries [24, 34], and Encryption[6]. In this paper we are specifically interested in detecting compilation errors [12] when compiling from a given language into WebAssembly. Finding faults in WebAssembly is an active research area[40], where previous work has looked at detecting side-channel attacks [31], visual function calls to aid in testing [41] and differential fuzzing [23], among others. Stiévenart *et al.* [45] first apply differential testing on WebAssembly to find security risks of porting C programs to WebAssembly. They rely on the Juliet Test Suite 1.3 [9], which contains 17,802 C programs with known flaws. Unlike this work, we start from constructing large numbers of valid testable C programs. We perform the first comprehensive analysis on potential misbehaviour when compiling to WebAssembly and give guidance to developers who plan to port C application to WebAssembly.

7 CONCLUSION

Transparent porting from C to WebAssembly face many difficulties in practice accomplish. Practitioners need to modify the application source code to make it fit the WebAssembly architecture. In this paper, we investigate what misbehavior will occur after porting C programs to WebAssembly. We collect 14,024 valid C programs from three sources, compile them into WebAssembly using Emscripten, and compare their execution output with the original program's. 61 of these programs exhibit different execution outputs. We manually analyze their root causes and divide them into nine root causes under three categories - "compiler", "runtime", and "language feature" . For each misbehavior type, we give advice to developers on how to handle them during practice. In conclusion, porting existing C programs to WebAssembly for cross-platform distribution still needs code adaptations, and more effort should be put into stitching inconsistencies between WebAssembly and other high-level languages.

REFERENCES

[1] 2021. Feature: WebAssembly Exception Handling. https://chromestatus.com/feature/4756734233018368. (2021).

[2] 2022. I Want to Compile a WebAssembly module from.... https://webassembly.org/getting-started/developers-guide/. (2022).

[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742–762.

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[5] Tapan Anand. 2020. Introducing Acrobat on the Web, Powered by WebAssembly. https://blog.developer.adobe.com/acrobat-on-the-web-powered-by-webassembly-782385e4947e.

[6] Nuttapong Attrapadung, Goichiro Hanaoka, Shigeo Mitsunari, Yusuke Sakai, Kana Shimizu, and Tadanori Teruya. 2018. Efficient two-level homomorphic encryption in prime-order bilinear groups and a fast implementation in webassembly. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 685–697.

[7] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*. IEEE, 207–212.

[8] Rick Battagline. 2019. *Hands-On Game Development with WebAssembly: Learn WebAssembly C++ programming by building a retro space game.* Packt Publishing Ltd.

[9] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[10] c testsuite. 2022. c-testsuite. https://github.com/c-testsuite/c-testsuite.

[11] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: a mutant generation tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1080–1084.

[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

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

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

[15] João De Macedo, Rui Abreu, Rui Pereira, and João Saraiva. 2022. WebAssembly versus JavaScript: Energy and Runtime Performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 24–34. https://doi.org/10.1109/ICT4S55073.2022.00014

[16] A. Denisov and S. Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 25–31. https://doi.org/10.1109/ICSTW.2018.00024

[17] Eduard P Enoiu, Daniel Sundmark, Adnan Čaušević, Robert Feldt, and Paul Pettersson. 2016. Mutation-based test generation for plc embedded software using model checking. In *IFIP International Conference on Testing Software and Systems*. Springer, 155–171.

[18] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552.

[19] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*. 147–158.

[20] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 621–631.

[21] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

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

[23] Gilang Hamidy et al. 2020. Differential Fuzzing the WebAssembly. (2020).

[24] Hunseop Jeong, Jinwoo Jeong, Sangyong Park, and Kwanghyuk Kim. 2018. WATT: A novel web-based toolkit to generate WebAssembly-based libraries and applications. In *2018 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 1–2.

[25] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[26] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM conference on Computer and communications security*. 279–288.

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

[28] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018.

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*. 1714–1730.

[29] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.

[30] Jacob Mattingley, Yang Wang, and Stephen Boyd. 2010. Code generation for receding horizon control. In *2010 IEEE International Symposium on Computer-Aided Control System Design*. IEEE, 985–992.

[31] Mohammad Erfan Mazaheri, Farhad Taheri, and Siavash Bayat Sarmadi. 2020. Lurking eyes: A method to detect side-channel attacks on javascript and webassembly. In *2020 17th International ISC Conference on Information Security and Cryptology (ISCISC)*. IEEE, 1–6.

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

[33] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*. 1–10.

[34] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2019. Gobi: WebAssembly as a practical path to library sandboxing. *arXiv preprint arXiv:1912.02285* (2019).

[35] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[36] Mike Papadakis, Nicos Malevris, and Maria Kallia. 2010. Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test*. 111–118.

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[38] Jamie Perkins. 2022. debugout.js. https://github.com/BeamNG/BeamNGpy.

[39] Mull Project. 2016-2022. Supported Mutation Operators. https://mull.readthedocs.io/en/0.17.0/SupportedMutations.html.

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

[41] Alan Romano and Weihang Wang. 2020. Wasmview: Visual testing for webassembly applications. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 13–16.

[42] Andreas Rossberg. 2022. WebAssembly Specification. https://webassembly.github.io/spec/core/index.html.

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

[44] Thomas Stei. 2022. Bringing Adobe's Creative Cloud to the web: Starting with Photoshop. https://www.youtube.com/watch?v=CF5zZZy0R9U. (2022).

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

[46] Surma. 2019. The main thread is overworked & underpaid (Chrome Dev Summit 2019). https://www.youtube.com/watch?v=7Rrv9qFMWNM&t=5s. (2019).

[47] GCC Team. 2022. GNU Compiler Collection. https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite/gcc.dg.

[48] W3C Team. 2015. Call for Participation in WebAssembly Community Group. https://www.w3.org/community/webassembly/2015/04/29/call-for-participation-in-webassembly-community-group/.

[49] WebAssembly Team. 2019. AutoCAD Web App. https://madewithwebassembly.com/showcase/autocad/.

[50] Nabeel Al-Shamma Thomas Nattestad. 2022. Photoshop's journey to the web. https://web.dev/ps-on-the-web/.

[51] Yu-Wen Tung and Wafa S Aldiwan. 2000. Automating test case generation for the new generation mission software system. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, Vol. 1. IEEE, 431–437.

[52] Luke Wagner. 2017. WebAssembly consensus and end of Browser Preview. https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html.

[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[54] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 665–676.