When Function Inlining Meets WebAssembly: Counterintuitive Impacts on Runtime Performance

Alan Romano
ajromano@usc.edu
University of Southern California
USA

Weihang Wang
weihangw@usc.edu
University of Southern California
USA

ABSTRACT

The WebAssembly standard defines a bytecode format serving as a compilation target for languages such as C, C++, and Rust. WebAssembly compilers are built on top of existing compiler infrastructures such as LLVM and newly developed compiler toolchains such as Binaryen, handling various new features of the WebAssembly language. However, we observe that both these new and existing infrastructures implicitly assume that the execution environments of native and WebAssembly applications are the same, ignoring the presence of browser compilers in the WebAssembly pipeline. This incorrect assumption often misguides function inlining optimizations, resulting in a slower WebAssembly module when function inlining is applied. This paper is the first to investigate the counterintuitive impacts of function inlining on WebAssembly runtime performance. We inspect the inlining optimization passes of the LLVM and Binaryen infrastructures used in the Emscripten C/C++-to-WebAssembly compiler. Our investigation on 127 C/C++ samples from the LLVM test suite shows that 66 samples exhibit counterintuitive behavior due to function inlining, particularly from inlining hot functions into long-running functions. We hope our findings motivate further work on revising existing optimizations with the unique characteristics of WebAssembly environments in mind.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Information systems** → **Web applications**.

KEYWORDS

WebAssembly, Function Inlining, LLVM, Binaryen, Emscripten

ACM Reference Format:
Alan Romano and Weihang Wang. 2023. When Function Inlining Meets WebAssembly: Counterintuitive Impacts on Runtime Performance. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616311

1 INTRODUCTION

WebAssembly (abbreviated Wasm) [34] is a low-level, statically typed language aiming to serve as a universal compilation target for the Web. It is designed to be fast to compile and run; to be portable, i.e., language-, hardware-, and platform-independent; and to have formal type and memory safety guarantees. WebAssembly is supported on all four major browsers (i.e., Chrome, Firefox, Safari, and Edge) [51] and compiles from several programming languages, including C, C++, C#, Rust, and Go [26]. Recent studies have shown that one out of every 600 websites use WebAssembly [35] for purposes such as games [40, 69], cryptography [60, 70], machine learning [66], and medical research [33, 38].

WebAssembly compilers leverage the same compiler infrastructures as compilers of traditional languages. For example, the Emscripten C/C++-to-WebAssembly compiler [10], the Rustc compiler [17], and Intel's oneAPI compiler [6] all use the LLVM [7] compiler infrastructure. Unfortunately, we observe that WebAssembly compilers leverage existing infrastructures without considering the differences between WebAssembly and native applications.

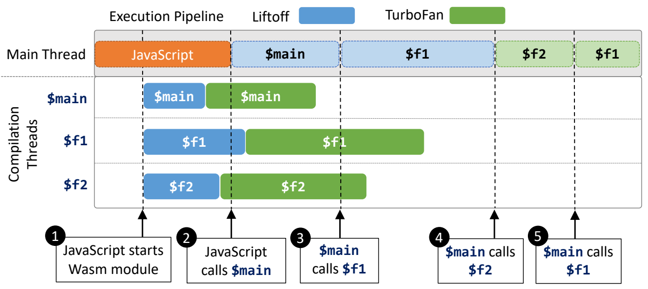


Figure 1: Chromium tier-up process. In this example, function `$main` **uses the Liftoff-generated code when first called as it is the only code available.** `$main` **calls** `$f1` **which only has Liftoff code ready.** `$f2` **uses the TurboFan-generated code as it is available at the first call. On the second call to** `$f1`**, its TurboFan-generated code is available and used for the call.**

One of the substantial differences is that WebAssembly has the *additional compilation layer at runtime* running within browsers, generating the final machine code for WebAssembly instructions. Browsers, such as Chromium [2] and Firefox [13], typically include *at least two* WebAssembly compilers: a fast compiler emitting unoptimized code and a slow compiler emitting highly optimized code. Browsers use both compilers to ensure the machine code for WebAssembly functions is available early and can perform faster once the optimized code is available. When the optimized code is ready, the code is *tiered-up* on the following function call invocation by replacing the unoptimized code with the optimized code. The tiering-up process only occurs on a function call because the

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616311

Table 1: Findings and Implications of Our Study.

	Findings	Implications
1	We identify counterintuitive function inlining behavior between WebAssembly's *compilation pipeline* and *execution pipeline*.	We show that function inlining slows WebAssembly runtime performance in some samples by as much as 15.5×.
2	We find that function inlining can introduce counterintuitive behavior in 51.97% (66/127) of the studied WebAssembly modules.	We investigate the characteristics of the inlined functions and find that larger code sizes can introduce the counterintuitive behavior.
3	We show that modifying Binaryen's inlining pass to avoid inlining hot functions reduces the counterintuitive behavior in 73.21% (41/56) of the modules.	This finding motivates further work on improving inlining heuristics for hot functions.
4	We are the first study to perform an in-depth investigation into the runtime impacts of a traditional optimization technique on WebAssembly binaries.	Our findings can motivate future work investigating the effects of other traditional compiler optimizations on WebAssembly modules.

unoptimized and optimized machine codes are not interchangeable. Figure 1 illustrates the Chromium tier-up process.

This tiering-up process complicates the effects of traditional optimization techniques such as *function inlining*, which moves function code into function call sites to reduce context switches. In doing so, function inlining reduces the number of call invocation sites for the tiering-up process to occur. This can cause an undesirable side effect: slowing down the runtime performance. Figure 2 shows that function inlining leads to worse runtime performance for samples within the LLVM test suite [22]. This figure compares the runtimes of the samples compiled to the *O3* optimization level with function inlining enabled to runtimes of the samples with function inlining disabled. The runtime can slow down by as much as 15.5×. In all the samples, function inlining decreases the number of call sites, leading to fewer opportunities for browsers to tier-up functions to more-optimized machine code.

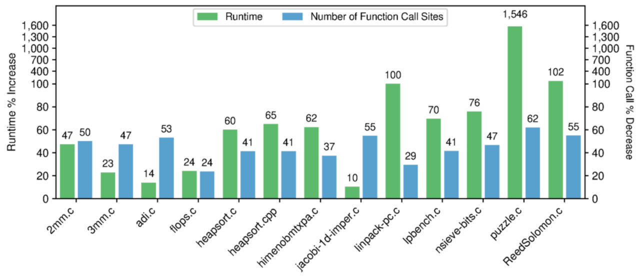


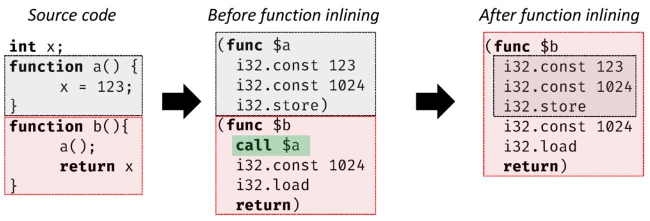
Figure 2: Function inlining slows runtime performance in Chromium. Using Emscripten with the *O3* optimization flag, the green bars show the % runtime speedup in the samples when function inlining is enabled compared to when inlining is disabled. The blue bars show the % decrease in the number of function call sites when inlining is enabled.

There has been much work studying compiler optimization techniques. Previous work has looked at the impacts of optimizations on specific platforms [28] and how optimizations affect SIMD performance [36]. Some works propose optimization selection strategies leveraging machine learning [41, 48]. While compiler optimizations are an active area of study, to the best of our knowledge, there are no systematic studies on the effects of function inlining in the WebAssembly compilation pipeline. Thus, we conduct the first empirical study on the impact of function inlining on WebAssembly runtime performance. We investigate function inlining used in Emscripten [10], a widely used C/C++ to WebAssembly compiler. We also inspect the inlining passes provided by two infrastructures used in Emscripten, LLVM [7] and Binaryen [18]. The findings and their implications of this study are listed in Table 1.

2 BACKGROUND

2.1 WebAssembly Compilation Pipeline

We illustrate the WebAssembly compilation pipeline using Emscripten [10], a compiler that converts C/C++ code to WebAssembly. Internally, Emscripten makes use of the Clang compiler [3] for its frontend and uses components from LLVM [7] and Binaryen [18] in its backend. Binaryen is a library providing tools to construct compiler infrastructures targeting WebAssembly. Binaryen defines an intermediate representation (IR) that closely models WebAssembly and enables WebAssembly-specific optimizations such as IR flattening to remove nested side effects and function reordering to shrink the encoding of the most called functions [18, 73].



```
int x;
function a() {
      x = 123;
}
function b(){
      a();
      return x
}
```

```
(func $a
  i32.const 123
  i32.const 1024
  i32.store)
(func $b
  call $a
  i32.const 1024
  i32.load
  return)
```

```
(func $b
  i32.const 123
  i32.const 1024
  i32.store
  i32.const 1024
  i32.load
  return)
```

Figure 3: Binaryen function inlining. In the original C code, function `b` calls function `a`. The WebAssembly code before inlining shows how both `a()` and `b()` are separate. After inlining, the instructions of `a()` are inlined into `b()`.

The Emscripten compilation pipeline begins by inputting the source code files into the Clang compiler. This compiler uses its *Frontend* component to parse the source files into an LLVM IR. The IR is passed to the *Middle-end* component, which implements several optimization passes, including the *inline* pass that performs function inlining. The pass moves instructions of a called function to the function call location, but first it checks to ensure that the inlined instructions can safely replace the call. For example, inlining indirect or external calls may break the program semantics. Additionally, the pass estimates the performance cost of inlining a function (e.g., a heuristic on the function's code size) to determine if it is beneficial to inline. The *Middle-end* component passes the optimized IR to the *CodeGen* component to create a WebAssembly module. Next, the module is passed to Binaryen's *wasm-opt* tool [18], which applies Binaryen's set of optimization passes to the module. In Binaryen, function inlining is performed by the *inlining-optimizing* pass. Similar to the *inline* pass in LLVM, the *inlining-optimizing* pass moves function instructions into the location of the original call site if the calculated inlining cost is less than a threshold value. Differences between these passes include the IR structures that are inlined as LLVM can also inline its block structures. Besides, Binaryen can support partial inlining of early-return conditional statements [18]. Figure 3 illustrates Binaryen's function inlining. Finally, the compilation pipeline outputs the optimized WebAssembly binary and JavaScript support code.

2.2 WebAssembly Execution Pipeline

The generated WebAssembly module and JavaScript files are run by a browser such as Chromium [2] or Firefox [54], which each have different internal compilers to generate machine code for the WebAssembly module. For example, Chromium is powered by the V8 JavaScript and WebAssembly engine [16], which includes two compilation engines to generate machine code for WebAssembly. The first compiler, *Liftoff* [25], is a single-pass compiler that emits machine instructions immediately after reading in a WebAssembly instruction at the expense of the number of optimizations that it applies. As a result, the Liftoff code can perform sub-optimally when executed. The second compiler, *TurboFan* [14], is a multi-pass compiler that applies several optimization passes to the machine code. While TurboFan generates faster code, this compiler takes much longer to generate code than Liftoff. To balance start-up speed with execution performance, Chromium first generates code for WebAssembly functions with Liftoff and immediately starts the TurboFan compilation. When the TurboFan code for a function is ready, the function code *tiers-up* by replacing the Liftoff code with the TurboFan code. Firefox uses the SpiderMonkey JavaScript and WebAssembly engine [13] to handle WebAssembly execution. Similar to V8, SpiderMonkey contains two compilation engines for WebAssembly. The first compiler, *Wasm-Baseline*, performs a fast translation of WebAssembly instructions to machine code for quick startup. The second engine, *Wasm-Ion*, applies optimizations on the emitted machine code. SpiderMonkey follows the tiering-up scheme by using Wasm-Baseline to emit machine code quickly while Wasm-Ion generates better-performing machine code.

3 COUNTERINTUITIVE INLINING EXAMPLE

We demonstrate how function inlining can counterintuitively impact runtime behavior using a sample benchmarking program, *random.cpp*, as an example. We present its source code and compiled WebAssembly code in Figure 4. We highlight the impact on two of the sample's functions when the function inlining is enabled and disabled. Figure 4(a) shows the C++ source code implementation of the functions `gen_random` and `main`. `gen_random` uses the constants `IM`, `IA`, and `IC` to generate a pseudo-random number. The `main` function calls `gen_random` in a long-running `while` loop performing 400 million iterations, making `gen_random` a hot function. Figure 4(b) shows the WebAssembly code of `wasm-function[13]` and `wasm-function[14]` when function inlining is disabled. The export section on line 180 shows that `wasm-function[13]` implements `main`. Inspecting the loop code within `wasm-function[13]` shows that it calls `wasm-function[14]` with the value 100.0 passed in as an argument, meaning that `wasm-function[14]` implements `gen_random`. Figure 4(c) shows the WebAssembly code for the `main` function, `wasm-function[13]`, produced when inlining is enabled. Inspecting Figure 4(b) and Figure 4(c) reveals that `wasm-function[14]` from Figure 4(b) has been inlined into `wasm-function[13]`.

When the Chromium browser runs this WebAssembly module, machine code for each function is first generated using the Liftoff compiler. Once this compiler finishes generating code for a function, the function can begin executing. In the background, the optimizing TurboFan compiler begins generating better-performing machine code for that function. When TurboFan finishes generating the machine code, the browser switches out the Liftoff-generated code for the TurboFan-generated code on the following function call. However, since `main` in a C program is only invoked once, the browser does not switch to the TurboFan-generated code. Because the hot function `gen_random` has been inlined into `main`, `gen_random` also uses the slower Liftoff code, and the program runtime performance is negatively impacted. This example shows how function inlining can cause counterintuitive runtime behavior in WebAssembly.

4 METHODOLOGY

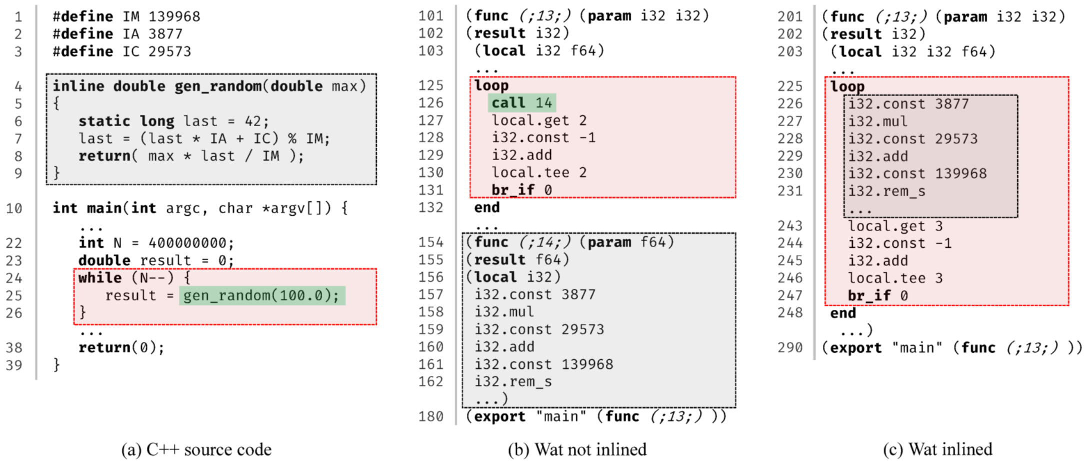
We aim to understand the counterintuitive effects of function inlining on WebAssembly program runtime. We define a *counterintuitive* effect as producing a binary with a slower runtime performance than if the optimization was disabled. Specifically, we focus on the following research questions:

- **RQ1 – Significance:** How often does function inlining counterintuitively impact WebAssembly modules, and are the effects unique to WebAssembly?
- **RQ2 – Function Characteristics:** Which characteristics of the inlined functions cause the counterintuitive behavior?
- **RQ3 – Quantification:** How does excluding certain functions from inlining impact the counterintuitive effects?

To answer these questions, we use samples from the LLVM test suite to perform five sets of experiments. Next, we discuss the C/C++ source programs and the experiments in detail.

4.1 C/C++ Source Programs

To measure the runtime performance impacts of different optimization configurations, we select 143 C/C++ samples totaling over 34,000 lines of code (LOC) from the LLVM test suite [8]. The test suite contains benchmarking samples measuring LLVM compilation performance. We focus on the samples within the *Single-Source/Benchmarks* directory, listed in Table 2, as these samples are designed to trigger optimizations and can be compiled by Emscripten without code changes. We select this test suite for its inclusion of samples used in prior works and it ease of compilation. This test suite includes samples from the Polybench benchmark suite [59], which was used by Jangda et al. to compare WebAssembly



```cpp
#define IM 139968
#define IA 3877
#define IC 29573

inline double gen_random(double max)
{
   static long last = 42;
   last = (last * IA + IC) % IM;
   return( max * last / IM );
}

int main(int argc, char *argv[]) {
   ...
   int N = 400000000;
   double result = 0;
   while (N--) {
      result = gen_random(100.0);
   }
   ...
   return(0);
}
```

(a) C++ source code

```
(func (;13;) (param i32 i32)
(result i32)
 (local i32 f64)
 ...
 loop
   call 14
   local.get 2
   i32.const -1
   i32.add
   local.tee 2
   br_if 0
 end
 ...
(func (;14;) (param f64)
(result f64)
(local i32)
 i32.const 3877
 i32.mul
 i32.const 29573
 i32.add
 i32.const 139968
 i32.rem_s
 ...)
(export "main" (func (;13;) ))
```

(b) Wat not inlined

```
(func (;13;) (param i32 i32)
(result i32)
 (local i32 i32 f64)
 ...
 loop
   i32.const 3877
   i32.mul
   i32.const 29573
   i32.add
   i32.const 139968
   i32.rem_s
   ...
   local.get 3
   i32.const -1
   i32.add
   local.tee 3
   br_if 0
 end
  ...)
(export "main" (func (;13;) ))
```

(c) Wat inlined

Figure 4: *random.cpp* C++ and WebAssembly code. (a) shows the C++ code of two functions, `gen_random` and `main`. (b) shows the WebAssembly output compiled with function inlining disabled. Function 13 is the `main` function, and inspecting its loop code reveals function 14 is `gen_random`. (c) shows function 13 from the module compiled in the Baseline experiment. The file differences between (b) and (c) show that function 14, `gen_random`, has been inlined into function 13, `main`.

Table 2: LLVM test suite *SingleSource/Benchmarks*.

Subdirectory	#Samples	#LOC	Subdirectory	#Samples	#LOC
Adobe-C++	6	696	Misc-C++	7	1,341
BenchmarkGame	8	549	Misc-C++-EH	1	16,846
CoyoteBench	4	1,294	Polybench	32	5,188
Dhrystone	2	767	Shootout	14	663
Linpack	1	552	Shootout-C++	25	972
McGill	4	956	SmallPT	1	102
Misc	27	3,487	Stanford	11	1,332
			Total	143	34,745

and native runtime [37]. The remaining samples implement samples of comparable computational complexity to those in the Polybench suite, such as Fibonacci number computation [20], Cholesky factorization [21], and Huffman compression [22]. In addition, the samples within the *SingleSource* directory of the test suite are designed so that only a single file is needed for compilation. The compilation settings needed to compile each sample are also well documented within the test suite repository. For these reasons, we choose the LLVM test suite samples for our experiments. We omit 16 samples that do not compile successfully with Emscripten, leaving 127 samples for our experiments.

4.2 Experiments Inspecting Inlining Effects

We start our investigation by establishing baseline runtime measurements using the four optimization level options, i.e., *O0*, *O1*, *O2*, and *O3* [11], available to an end user of the compiler. For our study, a binary is faster if its runtime is lower than that of another binary.

We describe the details of each optimization level below.

- *O0*: means no optimizations. The compiler compiles the source code without any attempt to optimize the code.
- *O1*: applies basic optimizations, such as loop simplification and redundant instruction combinations [9, 24].
- *O2*: adds more passes than *O1* while balancing between running time improvement and code size reduction.
- *O3*: contains all optimizations in *O2* and enables optimizations that increase code size to improve runtime.

To identify counterintuitive behavior caused by function inlining, we disable function inlining in each of the optimization levels and measure the runtime in each sample using both Chromium and Firefox. We focus on finding WebAssembly-specific issues surrounding inlining optimizations, so for every sample, we compile to the native x64 architecture and measure the runtime using the same optimization levels. Our analysis focuses on samples where the runtime behavior is intuitive on the native architecture and counterintuitive on WebAssembly. We then separately disable the LLVM *inline* and Binaryen *inlining-optimizing* passes in each optimization level to understand how each pass introduces the counterintuitive behavior. To do so, we construct modified builds of the LLVM and Binaryen that selectively disable passes. For the samples compiled to the native x64 architecture, we only disable the *inline* pass in LLVM as Binaryen is for WebAssembly only. We measure the sample runtimes when the inlining passes are enabled and disabled. The runtime behavior is counterintuitive if disabling the inlining pass results in a faster runtime than if the inlining pass is enabled.

To explore the causes of the counterintuitive behavior in the WebAssembly samples, we use the Linux *perf* tools to inspect the fine-grained runtime details of the execution. Specifically, we use the *perf record* tool to record which WebAssembly functions are executed and which browser compiler code they use. We also measure the overall percentage of execution time spent within these WebAssembly functions. We investigate the execution of native x64 samples using the *perf stat* tool. This tool records key hardware and software events, such as cache misses, branches taken, and CPU cycles, which allow us to understand the counterintuitive behavior.

Table 3: Experiments testing function inlining effects.

Experiment	Platform	Opt. Levels	LLVM Inlining Enabled	Binaryen Inlining Enabled
Baseline	Wasm	O0–O3	✓	✓
1	Wasm	O0–O3	✗	✗
2	x64	O0–O3	✗	N/A
3	Wasm	O0–O3	✗	✓
4	Wasm	O2–O3	✓	✗
5	Wasm	O2–O3	✓	Patched

```
 1 const origInstantiate = WebAssembly.instantiate;
 2 WebAssembly.instantiate = (source, importObject) => {
 3   let instance = origInstantiate(source, importObject);
 4   for(const name in instance.exports){
 5     const origFunction = instance.exports[name];
 6     instance.exports[name] = function(){
 7       const startTime = performance.now();
 8       let result = origFunction.apply(this, arguments);
 9       const endTime = performance.now();
10       logTime(name, startTime, endTime);
11       return result;
12 }}}
```

Figure 5: Instrumentation code to record runtime. The code modifies exported functions of all WebAssembly modules to log the time before and after the function call.

Table 3 presents the setups of our five experiments for inspecting the effects of function inlining. Similar to other studies [15, 49, 50, 55, 58], we use the average of 10 runs for all our experiments.

- *Baseline:* We measure the runtimes of the 127 benchmarking samples compiled to WebAssembly with the optimization levels *O0-O3* in their default modes, i.e., with both the LLVM *inline* and Binaryen *inlining-optimizing* passes enabled.
- *Experiment #1:* We compare the runtimes of all WebAssembly samples with *O0-O3* having the LLVM *inline* and Binaryen *inlining-optimizing* passes disabled.
- *Experiment #2:* We compile each sample to two versions of an x64 executable: one version with the LLVM *inline* pass enabled against another version with *inline* disabled.
- *Experiment #3:* To understand which inlining pass contributes to the counterintuitive behavior more, our third experiment compiles the 127 WebAssembly samples with only the LLVM *inline* pass disabled.
- *Experiment #4:* We compile the 127 samples with only the Binaryen *inlining-optimizing* pass disabled.
- *Experiment #5:* We inspect the samples that experience counterintuitive effects in Experiment #4. We patch the *inlining-optimizing* pass to selectively disable inlining for hot functions. The runtimes of samples using this patched pass are compared against those using the original pass.

5 IMPLEMENTATION

We instrument the WebAssembly APIs to measure the module runtime in the browser. Specifically, as shown in Figure 5, we instrument the `WebAssembly.instantiate` and `.instantiateStreaming` methods to log the duration of each exported function. This code snippet modifies the WebAssembly API methods (line 1) to iterate over all exported functions in the WebAssembly instance (line 4). A new function overwrites each exported function to record the time before (line 7) and after (line 9) calling the original function (line 8). The duration is logged for analysis after the page executes (line 10). We instrument the WebAssembly APIs using Puppeteer [12].

We build our performance measurement tool using Node.js (v14.18.2) [57]. We use the Emscripten compiler (v2.0.29) built on top of Clang (v14.0.0) to generate WebAssembly samples. In our experiments, we use Chromium (v99.0.4806.0) and Firefox (version 109.0). We run our Chromium and native experiments on a desktop containing an Intel Core i7 CPU@3.20GHz with 32 GB RAM. We run our Firefox experiments on a laptop with an Intel Core i7 CPU@2.10GHz and 64GB RAM. Both devices run Ubuntu 20.04.

6 EVALUATION

In this section, we present findings and insights from our five sets of experiments investigating the counterintuitive function inlining effects in WebAssembly modules.

Summary of Results. We find that disabling inlining in both LLVM and Binaryen causes counterintuitive behavior in 66 of the 127 (51.97%) samples run in Chromium and Firefox. Only 12 of these samples experience similar behavior on the native x64 platform, so we exclude these samples from our investigation[1]. We run two sets of experiments where one disables only LLVM's *inline* pass and the other disables only Binaryen's *inlining-optimizing* pass. LLVM's pass impacts 44 samples, while Binaryen's pass causes counterintuitive results in 56 samples. Further investigation of Binaryen's *inlining-optimizing* pass identifies hot functions as a probable cause of the counterintuitive effect. To quantify their impact, we modify the pass to prevent inlining for hot functions. 41 of the 56 samples experience improved runtime with the patched pass, indicating that hot functions inlined into long-running functions are a major cause of the counterintuitive behavior and further work should focus on improving inlining heuristics for hot functions.

6.1 RQ1: Significance of Counterintuitive Effects

6.1.1 Function Inlining Effects on WebAssembly. To measure the effects of function inlining, we compile 127 samples from the *SingleSource/Benchmarks* directory of the LLVM test suite with optimization levels *O0 - O3*. Experiment #1 compiles each sample with the function inlining passes in both LLVM (*inline)* and Binaryen (*inlining-optimizing*) disabled. The resulting modules are run using both Chromium and Firefox.

In Figure 6, we report the runtime speedup experienced by each sample when run in Chromium with the two inlining passes disabled. Note that the figure presents samples that contain at least one optimization level grouping that experience a 5% percent speedup, i.e., experience a 5% percent decrease in runtime, after disabling inlining. For example, in the sample *matrix.cpp*, disabling inlining leads to a runtime speedup of 62.42% in *O1* and 36.03% in *O2*. When compiled with inlining disabled, 32 samples become at least 5% faster (and 15 samples become 20% faster) in Chromium. Figure 7 shows that 51 samples with inlining disabled run at least 5% faster and 10 samples run at least 20% faster in Firefox. On average, these counterintuitive samples experience a 15.07% speedup. The results

[1]Such cases are *not* unique to WebAssembly, hence not our focus.

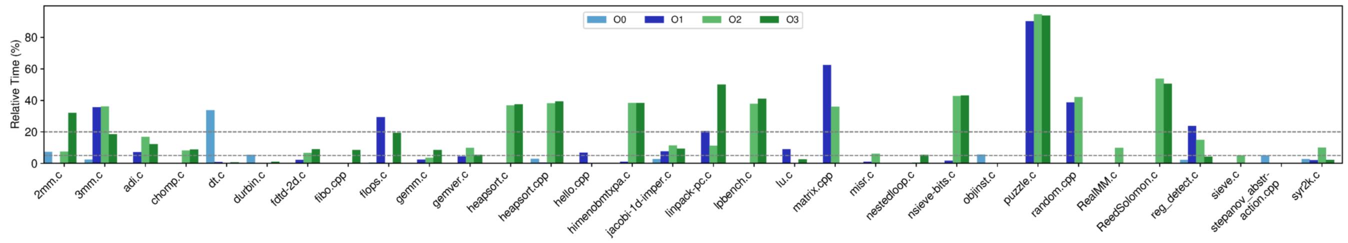


Figure 6: Runtime speedup of samples in Experiment #1 with Chromium. The bars show the % speedup of the samples having both LLVM and Binaryen inlining passes disabled compared to when both inlining passes enabled, i.e., the default version. The runtime speedups range from as low as 5.34% to high as 50.61% with an average speedup of 25.2%

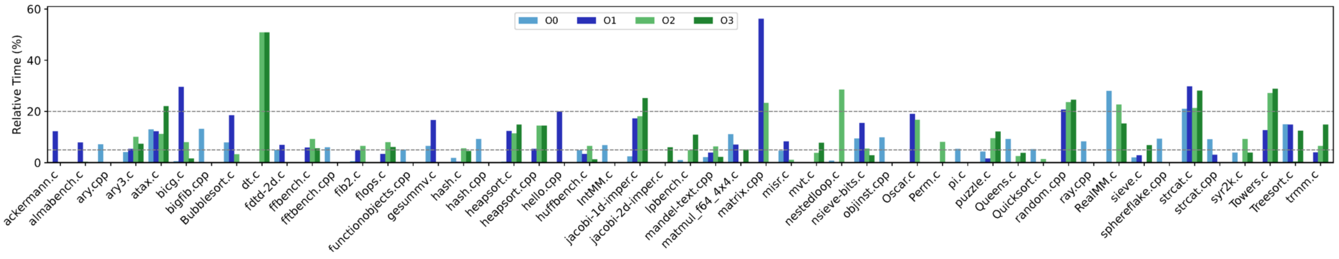


Figure 7: Runtime speedup of samples in Experiment #1 with Firefox. The runtime speedups range from as low as 5.02% in *functionobjects.cpp* to high as 56.15% in *matrix.cpp* with an average speedup of 15.07%

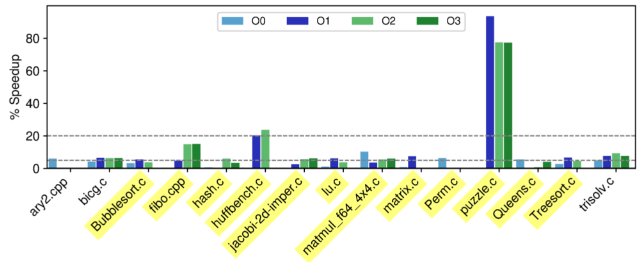


Figure 8: Runtime speedup of samples in Experiment #2. The bars show the % runtime speedup of the x64 samples after having LLVM's *inline* pass disabled.

in Figures 6 and 7 show function inlining causes counterintuitive behavior in certain WebAssembly modules.

6.1.2 Function Inlining Effects on x64 Architecture. We compare the effects of function inlining in WebAssembly against the inlining effects in native x64 binaries to determine whether these effects are unique to WebAssembly or common across multiple platforms. Experiment #2 compiles the samples to x64 binaries with *inline* in LLVM disabled. Figure 8 shows the samples experiencing the counterintuitive inlining behavior. Of the 127 samples, 15 samples experience similar counterintuitive effects with function inlining as those in WebAssembly. Of those 15 samples, 12 samples, highlighted in yellow, experience counterintuitive inlining effects on both WebAssembly and the native x64 platform. We inspect the execution details of the native binaries using the *perf stat* tool. We find that higher values in the `cache-misses`, `all-loads-retired`, and `all-stores-retired` events explain the counterintuitive behavior experienced by these samples. These execution statistics indicate that inlining leads to an increased number of cache misses (from the `cache-misses` events) and increased register pressure (from the `all-loads-retired` and `all-stores-retired` events) [37] during execution. These are known issues of function inlining in native architectures [5]. Among the samples experiencing increased counts in these metrics, inlining increases `cache-misses` by 12.95%, `all-loads-retired` by 11%, and `all-stores-retired` by 9%. The small number of common samples in both WebAssembly and native x64 suggests that the counterintuitive effects seen in the WebAssembly samples are caused by different factors than in the native x64 platform. We continue our investigation of the counterintuitive WebAssembly samples to determine what these factors are.

6.2 RQ2: Investigation of Function Characteristics Causing Counterintuitive Effects

The results in Section 6.1 show that the function inlining passes in Binaryen and LLVM can lead to counterintuitive behavior that is unique to WebAssembly. In this section, we investigate the causes of this behavior. Specifically, we seek to understand the characteristics of the inlined functions that lead to the counterintuitive effects we observe. We first manually inspect the compiled WebAssembly modules to understand how the code differences produce the behavior. However, the number of differences between the samples with both inlining passes enabled and disabled is large. Each sample contains an average difference of 24,774.73 LOC between the versions having inlining enabled and disabled. Combined with the terse syntax of WebAssembly, manual inspection of these samples proves to be extremely challenging. Hence, to ease the inspection process, we inspect the effects of *a single inlining pass* on the module, reducing the search space of code differences between the samples with enabled and disabled inlining. We also compile the samples with the LLVM and Binaryen inlining passes disabled separately to understand how each component introduces counterintuitive behavior and discuss the results of each component separately.

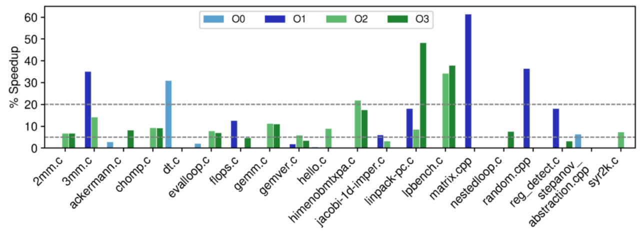


Figure 9: Runtime speedup comparing Experiment #3 with Baseline in Chromium. The bars show the % speedup in the samples’ runtimes with the LLVM *inline* pass disabled compared to their runtime with the *inline* pass enabled.

6.2.1 Function Inlining in LLVM. Experiment #3 aims to show the impact of the LLVM *inline* pass on the counterintuitive effects. The results, shown in Figure 9, reveal that the disabling the *inline* pass leads to a Chromium runtime speedup of at least 5% in 20 different samples with the average runtime speedup being 17.61%. Of those 20 samples, 7 samples experience a runtime speedup of at least 20%. In Firefox, 32 and 7 samples experience runtime speedups of at least 5% and 20%, respectively. The runtime speedups in Firefox range from as low as 5.03% to high as 56.80% with an average speedup of 13.01%. Due to space constraints, we omit the Firefox results figure.

Case Study (random.cpp). To exemplify the characteristics of the inlined functions, we investigate one of these affected samples, *random.cpp*, in depth. Recall that in its source and WebAssembly code presented in Figure 4, we saw that the code for the hot C++ function `gen_random` (`wasm-function[14]` in WebAssembly) is inlined into the C++ `main` function (`wasm-function[13]`). In this example, inlining is performed by LLVM’s *inline* pass Figure 10 shows the collected browser call trace from *perf record* for *random.cpp* when the *inline* pass is enabled and disabled. Figure 10(a) shows that when *inline* is enabled, `wasm-function[13]` consumes the largest WebAssembly runtime, i.e., it is the hottest function, and uses the Liftoff-generated code. In Figure 10(b), `wasm-function[14]` supplants `wasm-function[13]` as the function with the largest share of the overhead. `Wasm-function[14]` uses the TurboFan-generated code while `wasm-function[13]` uses the Liftoff code. Combining the code snippet in Figure 4(c) with the call trace in Figure 10(a) shows that when inlining is enabled, the browser is stuck using the Liftoff-generated code of `gen_random`. Figure 4(b) and Figure 10(b) show that disabling inlining allows `gen_random` to use the TurboFan code because it is not bound to the `main` function. To summarize, separating the hot `gen_random` function from the `main` function allows the hot function to use the more-efficient TurboFan code.

The *random.cpp* case shows that the *inline* pass affects whether the hottest function in the module uses the code generated by the Liftoff or TurboFan compiler. We collect the *perf record* output of each sample in Figure 9 with inlining enabled and disabled. We find that in 12 out of 20 Chromium samples, the hottest function of the module uses Liftoff when inlining is enabled and TurboFan when inlining is disabled. This finding suggests that function inlining can prevent the more performant TurboFan code from being used.

```
67.25%  [.] Function:wasm-function[13]-13-liftoff
 0.00%  [.] Function:wasm-function[1476]-1476-turbofan
 0.00%  [.] Function:wasm-function[1477]-1477-turbofan
 0.00%  [.] Function:wasm-function[285]-285-turbofan
 0.00%  [.] Function:wasm-function[327]-327-turbofan
```

(a) *Inline* Enabled

```
42.08%  [.] Function:wasm-function[14]-14-turbofan
11.06%  [.] Function:wasm-function[13]-13-liftoff
 0.00%  [.] Function:wasm-function[12]-12-liftoff
 0.00%  [.] Function:wasm-function[1507]-1507-turbofan
 0.00%  [.] Function:wasm-function[1510]-1510-liftoff
 0.00%  [.] Function:wasm-function[1524]-1524-liftoff
```

(b) *Inline* Disabled

Figure 10: *Perf record* output for *random.cpp*. (a) traces the browser function calls made during the Baseline experiment. Here, function 13 uses Liftoff code and occupies most of the execution time. (b) traces the browser calls in Experiment #3, and it shows that function 14 uses its TurboFan code.

6.2.2 Function Inlining in Binaryen. Experiment #4 aims to highlight the contribution that *inlining-optimizing* pass in Binaryen has on the counterintuitive behavior. Figure 11 shows that the *inlining-optimizing* pass in Binaryen causes 29 samples to experience counterintuitive runtime behavior in Chromium. Among these 29 samples, the smallest runtime speedup was 5.60% in *reg_detect.c*, the largest speedup was 50.0% in *ReedSolomon.c*, and the average speedup was 20.88%. It is important to note that since Experiment #4 inspects all 127 samples, these 29 samples are not a subset of the samples in Experiment #1. In Firefox, Figure 12 shows that 40 samples experience counterintuitive runtime speedup of at least 5%, with the smallest runtime speedup being 5.13% (*dry.c*) and the largest speedup being 42.39% (*lists.cpp*).

Our analysis of the affected samples with *perf record* shows that they experience similar behavior as *random.cpp* exhibits with LLVM’s *inline* pass (shown in Figure 10). In 23 of the 29 Chromium samples, the hottest function in the sample version with *inlining-optimizing* enabled uses the Liftoff-generated code, while the hottest function in the sample version with *inlining-optimizing* disabled uses the TurboFan-generated code. This finding shows that, for both components, the same reason explains the effects of function inlining: function inlining can prevent the hot functions in a module from using the more performant browser compiler.

Figure 13 shows the breakdown of executed functions within the Chromium-run *O2* samples in Figure 6 that have a speedup of at least 20%. For each sample, the figure shows two bars: the left bar is breakdown of the function executed when both inlining passes are enabled (Baseline), while the right bar shows the breakdown when inlining is disabled (Experiment #1). Each segment represents a single function, and its height indicates its percentage of the total execution time (according to *perf record*). As Figure 13 shows, the

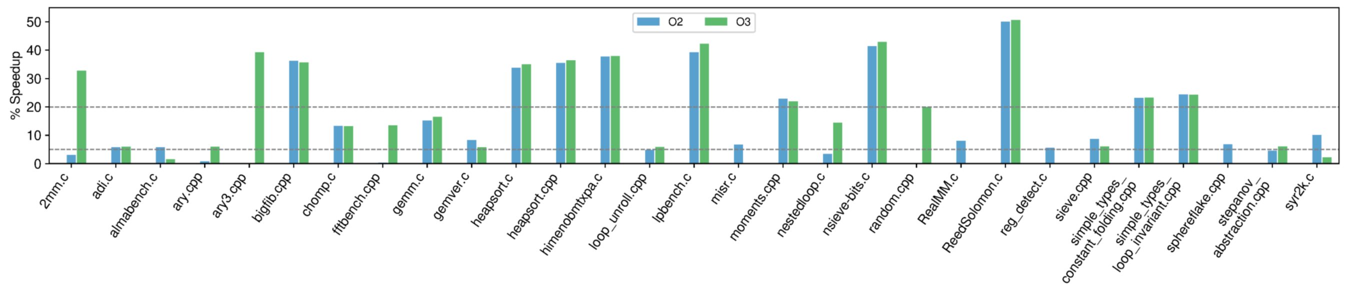


Figure 11: Runtime speedup comparing Experiment #4 with Baseline in Chromium. The bars present the runtime speedup in samples after the Binaryen *inlining-optimizing* pass is disabled.

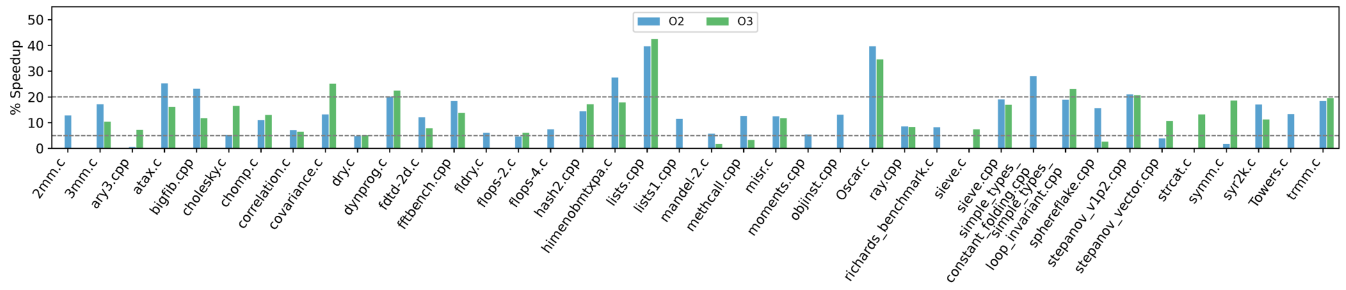


Figure 12: Runtime speedup comparing Experiment #4 with Baseline in Firefox.

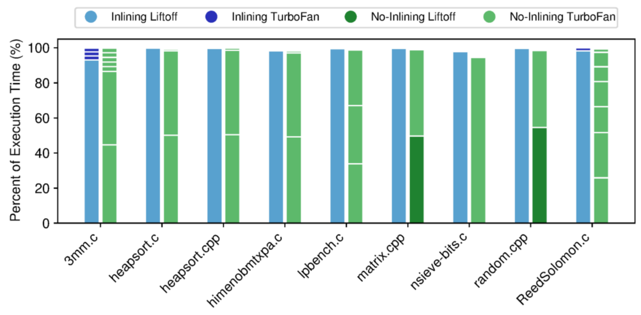


Figure 13: Breakdown of Liftoff or TurboFan compilers used in *O2*. Each bar segment represents a function and its portion of total execution time. The left bar shows the Baseline execution, while the right bar shows Experiment #1 execution.

```
8.98% [.] Function:wasm-function[7]-7-liftoff
0.14% [.] Function:wasm-function[5]-5-turbofan
0.00% [.] Function:wasm-function[10]-10-turbofan
0.00% [.] Function:wasm-function[21]-21-turbofan
0.00% [.] Function:wasm-function[22]-22-turbofan
0.00% [.] Function:wasm-function[23]-23-turbofan
0.00% [.] Function:wasm-function[27]-27-turbofan
0.00% [.] Function:wasm-function[6]-6-turbofan
0.00% [.] Function:wasm-to-js:iiii:i-0-turbofan
```

(a) *Inlining-Optimizing* Enabled

```
9.23% [.] Function:wasm-function[7]-7-turbofan
0.15% [.] Function:wasm-function[8]-8-turbofan
0.15% [.] Function:wasm-function[20]-20-turbofan
0.00% [.] Function:wasm-function[10]-10-turbofan
0.00% [.] Function:wasm-function[13]-13-turbofan
0.00% [.] Function:wasm-function[29]-29-turbofan
0.00% [.] Function:wasm-function[31]-31-turbofan
0.00% [.] Function:wasm-function[32]-32-turbofan
0.00% [.] Function:wasm-function[33]-33-turbofan
0.00% [.] Function:wasm-function[40]-40-turbofan
0.00% [.] Function:wasm-function[41]-41-turbofan
0.00% [.] Function:wasm-function[9]-9-turbofan
```

(b) *Inlining-Optimizing* Disabled

Figure 14: *Perf record* output for *Chomp.c* using O2

samples experience similar counterintuitive behavior to Figure 10: the samples are limited to using Liftoff for the hot functions when inlined into a long-running function, while disabling inlining allows the hot functions to use the TurboFan code.

Examples (chomp.c and Simple_types_constant_folding.cpp). Figure 14 shows the result of *perf record* when run on the sample *chomp.c*, with Figure 14a showing the output when *inlining-optimizing* is enabled and Figure 14b showing the output when *inlining-optimizing* is disabled. In both snippets, `wasm-function[7]` is the WebAssembly function with the largest portion of the execution time. However, Figure 14a shows that this function is executed with the Liftoff-generated machine code, while Figure 14b shows that the function uses the TurboFan-generated code.

Similarly, Figure 15 shows that the hottest function in Figure 15a, `wasm-function[7]`, using the Liftoff-generated code while the hottest function in Figure 15b, `wasm-function[120]`, manages to use the TurboFan code. Liftoff generates less-optimized, less-performant

```
8.23% [.] Function:wasm-function[7]-7-liftoff
2.74% [.] Function:wasm-function[10]-10-turbofan
0.75% [.] Function:wasm-function[34]-34-turbofan
0.60% [.] Function:wasm-function[35]-35-turbofan
0.11% [.] Function:wasm-function[20]-20-turbofan
0.08% [.] Function:wasm-function[22]-22-turbofan
0.08% [.] Function:wasm-function[25]-25-turbofan
0.08% [.] Function:wasm-function[31]-31-turbofan
```

(a) *Inlining-Optimizing* Enabled

```
1.05% [.] Function:wasm-function[120]-120-turbofan
0.89% [.] Function:wasm-function[135]-135-turbofan
0.46% [.] Function:wasm-function[121]-121-turbofan
0.45% [.] Function:wasm-function[122]-122-turbofan
0.45% [.] Function:wasm-function[137]-137-turbofan
                        ...
0.11% [.] Function:wasm-function[146]-146-turbofan
0.11% [.] Function:wasm-function[126]-126-turbofan
0.11% [.] Function:wasm-function[133]-133-turbofan
0.11% [.] Function:wasm-function[140]-140-turbofan
0.00% [.] Function:wasm-function[7]-7-liftoff
```

(b) *Inlining-Optimizing* Disabled

Figure 15: *Perf record* output for *Simple_types_constant_folding.cpp* using O2

code than TurboFan, so this difference caused by the *inlining-optimizing* pass means that the hottest function will be limited in performance. We can also see at the bottom of the list in Figure 15b that `wasm-function[7]` still uses the Liftoff-generated code. However, since this function no longer contains the hot code, the detrimental runtime impact is mitigated. In both samples, we can see that they experience the same runtime issues, the inability to switch their hottest function over to the TurboFan-generated code, that are seen in Figure 10 with LLVM's *inline* pass. This behavior shows that the same underlying issue, the hottest code being run with the slower Liftoff-generated code, affects both inlining passes across the different infrastructures.

The timing of the switch between Liftoff and TurboFan is determined by each compiler's duration for a given function. The architecture of the single-pass Liftoff compiler [25] along with the graph construction used in the TurboFan compiler [53] suggest that function code size should influence the compiler duration. We plot the compilation duration of the TurboFan and Liftoff compilers against the function code size of all functions from the counterintuitive samples in Figure 16. Our results reveal that the duration of both compilers closely follow the increase in code size. However, the TurboFan duration is consistently an order of magnitude larger than the Liftoff duration for the same function size.

Following this trend, an explanation on why Binaryen's *inlining-optimizing* pass and LLVM's *inline* pass lead to counterintuitive behavior is that inlining instructions to a call site increases the function code size. By increasing the code size of the long-running function, it quickly increases the compilation time of TurboFan and forces any inlined hot functionality to spend more time using the Liftoff code. In the case of a long-running function invoked only once, the increase in compilation duration decreases the likelihood that the TurboFan code will be ready by the time of the first, and only, invocation. Also, inlining hot functions into a function that is only invoked once, such as `main`, can prevent the inlined code from ever using the more-efficient TurboFan code.

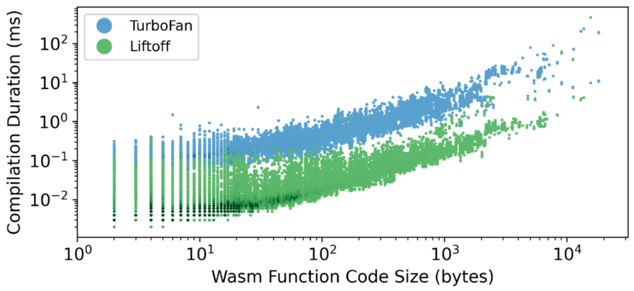


Figure 16: Chromium compilation duration among all samples. For each function, the TurboFan (blue dots) and Liftoff (green dots) compiler duration is plotted against the code size. Darker dots indicate multiple functions of the same size also have the same duration.

6.3 RQ3: Impact of Hot Functions on Counterintuitive Effects

In this section, we empirically quantify the impact of inlining hot functions in our dataset to better understand *potential performance gain* if the hot function inlining is disabled. Specifically, we modify Binaryen's *inlining-optimizing* in the compiler to exclude hot functions (i.e., functions called in loops) from being inlined. Then, we repeat the experiments with the modified pass to measure the performance improvement by the modification.

Identifying Hot and Long-running Functions. To identify possible hot functions, we search for functions called within a loop as they will be called repeatedly. Specifically, we use the LLVM parser tools [4, 27] to obtain the abstract syntax tree (AST) of the source code. We then traverse the AST to identify a sub tree with a root node of `For_Stmt` containing a `Call_Expr_Stmt` node. Such a subtree represents a function call statement within a loop and the `Call_Expr_Stmt`'s call target is a hot function. While it is not a comprehensive method, we find that this criterion is suitable for identifying the hot functions within our samples. Note that we manually verified that all the identified functions are hot functions. The list of functions matching this criterion is used to prevent them from being inlined. We also exclude typical long-running functions that only execute once, such as the entry point (e.g., `main`) function.

Patched inlining-optimizing Pass. In Section 6.2, we discuss how Binaryen's *inlining-optimizing* pass can prevent hot functions from tiering up to the more-efficient TurboFan code. Hence, in this experiment, we modify the *inlining-optimizing* pass to disable the inlining optimization for identified hot and long-running functions.

Results. We use this patched *inlining-optimizing* pass on the samples that experience the counterintuitive behavior in Figures 11 and 12. We list the change in original runtime versus the runtime with the modified Binaryen pass in Figures 17 and 18. We also list the original runtime versus the runtime results with *inlining-optimizing* disabled to understand the impact of inlining optimizations except for the hot functions. We find that, in 41 of the 56 samples from both browsers, excluding the hot functions from the inlining leads to improved runtime performance. Figure 17 shows that the runtime

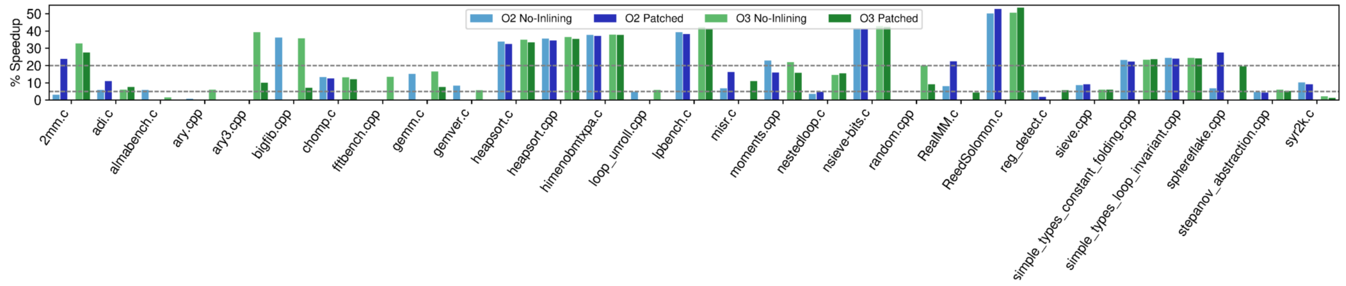


Figure 17: Runtime speedup of Experiments #4 & #5 against Baseline in Chromium. The dark blue and green bars represent the % speedup of the sample when compiled with our patched Binaryen pass for *O2* and *O3*, respectively. The light blue and green bars show the % speedup from Experiment #4 to serve as a reference point on the patched runtime impact. In 24 samples, our patch produces a speedup to a similar extent that disabling all inlining does.

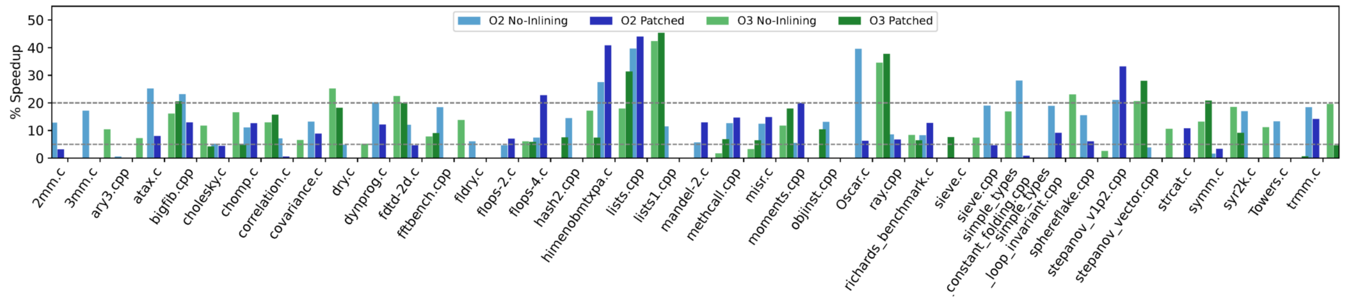


Figure 18: Runtime speedup of Experiments #4 & #5 against Baseline in Firefox.

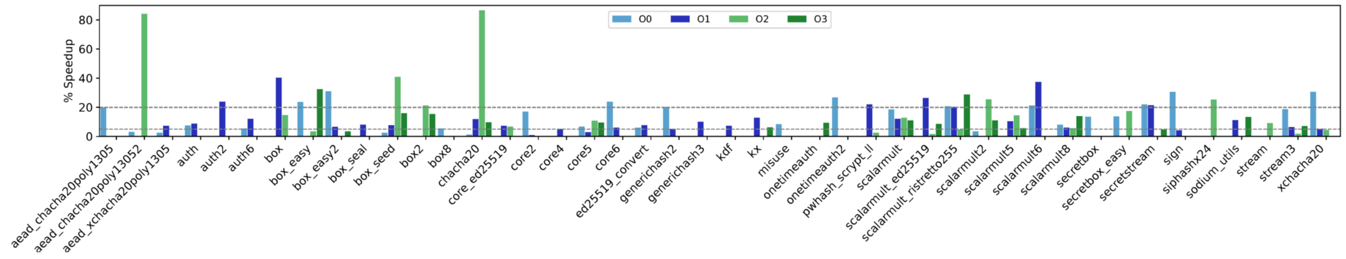


Figure 19: Libsodium.js runtime speedup of Experiment #1 in Chromium.

speedups of the 24 improved samples in the 29 Chromium samples range from as low as 5.24% in *stepanov_abstraction.cpp* to as high as 53.49% in *ReedSolomon.c*, with an average speedup of 22.58%. Figure 18 shows that the runtime speedups of the 24 improved samples among the 40 Firefox samples range from as low as 5.85% in *flops-2.c* to as high as 45.39% in *lists.cpp*, with an average speedup of 16.16%. It is important to note that our simple heuristic for determining hot functions is not comprehensive. Nevertheless, we show that inlining heuristics following this direction can improve runtime performance. We hope our findings motivate the need for re-examined inlining heuristics for WebAssembly compilation.

6.4 Case Study on a Real-World Application

We now investigate the effects of function inlining in one popular real-world WebAssembly application, Libsodium.js [31]. This project ports the Sodium cryptographic library [31] to the web, and it has over 396,000 weekly downloads on the NPM package registry [19]. This library provides APIs that implement cryptographic functions, e.g., encryption, message signing, and hashing. In addition, we choose this project for our case study as the project includes its C/C++ source code and compilation scripts. It contains 75 example programs that test the cryptographic APIs provided by the library, and we replicate Experiment #1 on these 75 programs. Figure 19 shows that function inlining from the LLVM and Binaryen passes causes counterintuitive behavior in 44 programs on Chromium. The smallest runtime speedup was 5.07% in *generichash2*, the largest speedup was 86.46% in *chacha20*, and the average speedup was 16.76%. In Firefox, we find that 58 programs show similar behavior, and the average speedup is 20.36%. These results show that this inlining issue extends to real-world applications.

7 DISCUSSION

7.1 Limitations

Our investigation of WebAssembly performance suffers from two main limitations. First, the precision of our custom-built JavaScript measurement tool limits the depth of our investigation. Most browsers limit JavaScript timers to millisecond resolution [52], which is too coarse to measure a typical WebAssembly call. As a result, we focus on samples that have long running functions with runtimes in the magnitude of seconds. We also focus on samples with a percent decrease greater than 5% to account for the lack of precision.

Our second limitation is that we only inspect two browsers, Chromium and Firefox. Inspecting each browser adds additional manual work, and we are limited by our budget of manual effort available. We accept this limitation as Chromium-based browsers and Firefox account for 74% of the browser market share [1].

7.2 Threats to Validity

7.2.1 Internal Validity. Our study results are subject to possible errors in the manual inspection processes. We manually inspect the emitted code to ensure that function inlining is present or omitted as per the tested configuration. We use the average of 10 runs to ensure changes are not caused by small runtime variations. Multiple factors, such as hardware, operating system, and system load, make it difficult to reproduce the exact runtime values we record. However, we describe the steps used to establish our Baseline experiment. The counterintuitive behavior, relative to the baseline, should remain consistent across different experimental setups.

7.2.2 External Validity. We use benchmarking samples from the LLVM test suite. As Emscripten is an LLVM-based compiler, we find that this collection of benchmarks curated by the LLVM development team is well-suited to assess the compilation effects caused by the inlining passes. The compiler benchmark samples also perform intensive computations, an intended use case of WebAssembly.

7.2.3 Construct Validity. We identify the runtime impacts of function inlining optimizations by measuring the program runtime through browser execution timing, native execution timing, and event profiling tools. These measurement methods should highlight changes caused by different optimizations used in the samples.

7.3 Future Work

Our current work only investigates the counterintuitive behavior of two inlining optimization passes. Our measurements show cases where these two passes alone cannot explain the counterintuitive behavior, indicating that other optimization passes also cause the behavior. We plan to study the other LLVM and Binaryen passes for similar counterintuitive behavior.

Our current analysis only focuses on a single metric for counterintuitive behavior: runtime performance. We plan to investigate possible counterintuitive changes in other metrics, such as code size, memory usage, and energy consumption.

8 RELATED WORK

Compiler Optimizations. Existing work studies the impacts of different optimizations on specific processor architectures [28] and high-level synthesis [30]. Some work proposes optimization frameworks improving SIMD performance [36]. Other works leverage machine learning techniques on optimization selection [41, 48]. Theodoridis et al. describe LLVM inlining heuristics improvements in native applications [68]. To our knowledge, our work is the first to study inlining performance in WebAssembly compilers.

Compiler Studies. Previous compiler studies investigate the prevalence of compiler bugs [62, 67] and survey different compiler testing approaches [29]. Other studies develop compiler testing techniques, such as equivalence modulo inputs [42, 43] and skeletal program enumeration [74].

WebAssembly Performance Measurements. Yan et al. [72] find evidence of optimizations causing counterintuitive effects. Jangda et al. [37] compare the performance of C programs compiled to WebAssembly and native code. In contrast, our work focuses on effects of function inlining on WebAssembly applications.

WebAssembly Program Analysis and Security. Several works analyze WebAssembly execution and security. Hilbig et al. [35] report the use cases and statistics of real-world WebAssembly binaries. Several tools dynamically analyze WebAssembly execution [45, 64], identify module purposes [63], and recover high-level type information from the binaries [46]. Prior work proposes specification and compiler extensions to improve security [32, 39, 56, 70, 71]. Other works identify vulnerabilities in WebAssembly applications [44, 47], propose attack strategies using WebAssembly [61], and detect malicious WebAssembly modules [65].

9 CONCLUSION

Function inlining optimizations in WebAssembly compilers fail to consider the presence of multiple browser compilers, leading to runtime performance issues. We provide the first in-depth investigation on the counterintuitive impact that function inlining can have on WebAssembly modules. Inlining can prevent hot functionality in the modules from leveraging optimized machine code if the functions are inlined into long-running or seldomly invoked functions, leading to noticeable performance degradation of the whole application. We find that this behavior effects 66 out of 127 samples in the LLVM test suite and is caused by the inlining passes in both the LLVM and Binaryen components of Emscripten. We hope our work highlights the need to revisit existing optimization techniques for optimal WebAssembly usage.

10 DATA AVAILABILITY

We make our experiment results and data collection scripts available on Zenodo at https://zenodo.org/record/7041455 [23]. This artifact contains the measured runtime results for all of our experiments and the scripts used to run the experiments.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was partially supported by the US National Science Foundation under Grant No. 2047980. 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] [n. d.]. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share.
[2] [n. d.]. Chromium. https://www.chromium.org/Home/.
[3] [n. d.]. Clang C Language Family Frontend for LLVM. https://clang.llvm.org/.
[4] [n. d.]. Clang.Cindex — Libclang 14.0.6 Documentation. https://libclang.readthedocs.io/en/latest/_modules/clang/cindex.html.
[5] [n. d.]. Inline Functions, C++ FAQ. https://isocpp.org/wiki/faq/inline-functions#inline-and-perf.
[6] [n. d.]. Intel C/C++ Compilers Complete Adoption of LLVM. https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html.
[7] [n. d.]. The LLVM Compiler Infrastructure Project. https://llvm.org/.
[8] [n. d.]. Llvm-Test-Suite/Matrix.c at Main · Llvm/Llvm-Test-Suite. https://github.com/llvm/llvm-test-suite.
[9] [n. d.]. LLVM's Analysis and Transform Passes — LLVM 13 Documentation. https://releases.llvm.org/13.0.0/docs/Passes.html#argpromotion-promote-by-reference-arguments-to-scalars.
[10] [n. d.]. Main — Emscripten 3.1.1-Git (Dev) Documentation. https://emscripten.org/.
[11] [n. d.]. Optimizing Code — Emscripten 3.1.6-Git (Dev) Documentation. https://emscripten.org/docs/optimizing/Optimizing-Code.html.
[12] [n. d.]. Puppeteer | Tools for Web Developers. https://developers.google.com/web/tools/puppeteer.
[13] [n. d.]. SpiderMonkey — Firefox Source Docs Documentation. https://firefox-source-docs.mozilla.org/js/index.html.
[14] [n. d.]. TurboFan · V8. https://v8.dev/docs/turbofan.
[15] [n. d.]. Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies: ACM SIGPLAN Notices: Vol 50, No 2. https://dl.acm.org/doi/abs/10.1145/2775052.2661090.
[16] [n. d.]. V8 JavaScript Engine. https://v8.dev/.
[17] [n. d.]. What Is Rustc? - The Rustc Book. https://doc.rust-lang.org/rustc/index.html.
[18] 2022. Binaryen. WebAssembly.
[19] 2022. Libsodium. https://www.npmjs.com/package/libsodium.
[20] 2022. Llvm/Llvm-Test-Suite/SingleSource/Benchmarks/Misc-C++/Bigfib.Cpp. LLVM.
[21] 2022. Llvm/Llvm-Test-Suite/SingleSource/Benchmarks/Misc-C++/Huffbench.c. LLVM.
[22] 2022. Llvm/Llvm-Test-Suite/SingleSource/Benchmarks/Polybench/Linear-Algebra/Kernels/Cholesky/Cholesky.c. LLVM.
[23] Alan and Weihang. [n. d.]. *Dataset for "When Function Inlining Meets WebAssembly: A Counterintuitive Effect on Runtime Performance"*. https://doi.org/10.5281/zenodo.7041455
[24] Antoine. 2013. Answer to "Clang Optimization Levels".
[25] Clemens Backes. [n. d.]. Liftoff: A New Baseline Compiler for WebAssembly in V8 · V8. https://v8.dev/blog/liftoff.
[26] Matteo Basso. 2022. Awesome Wasm.
[27] Eli Bendersky. [n. d.]. Parsing C++ in Python with Clang. https://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang.
[28] Aart JC Bik, David L Kreitzer, and Xinmin Tian. 2008. A case study on compiler optimizations for the Intel® Core TM 2 Duo Processor. *International Journal of Parallel Programming* 36, 6 (2008), 571–591.
[29] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (Feb. 2020), 4:1–4:36. https://doi.org/10.1145/3363562
[30] Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang. 2012. A study on the impact of compiler optimizations on high-level synthesis. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 143–157.
[31] Frank Denis. 2023. Libsodium.Js.
[32] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3337167.3337171
[33] Richard Finney and Daoud Meerzaman. 2018. Chromatic: WebAssembly-Based Cancer Genome Viewer. *Cancer Informatics* 17 (Jan. 2018), 1176935118771972. https://doi.org/10.1177/1176935118771972
[34] 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.
[35] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021 (WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138
[36] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. 2009. A SIMD optimization framework for retargetable compilers. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 1 (2009), 1–27.
[37] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: analyzing the performance of webassembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 107–120.
[38] Sébastien Jodogne. 2018. The Orthanc Ecosystem for Medical Imaging. *Journal of Digital Imaging* 31, 3 (June 2018), 341–352. https://doi.org/10.1007/s10278-018-0082-y
[39] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. , : SFI safety for native-compiled Wasm. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. https://doi.org/10.14722/ndss.2021.24078
[40] Jukka Jylänki. [n. d.]. WebAssembly for Native Games on the Web – Mozilla Hacks - the Web Developer Blog. https://hacks.mozilla.org/2017/07/webassembly-for-native-games-on-the-web.
[41] Yuriy Kashnikov, Jean Christophe Beyler, and William Jalby. 2012. Compiler optimizations: Machine learning versus o3. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 32–45.
[42] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *ACM SIGPLAN Notices* 49, 6 (June 2014), 216–226. https://doi.org/10.1145/2666356.2594334
[43] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. *ACM SIGPLAN Notices* 50, 10 (Oct. 2015), 386–399. https://doi.org/10.1145/2858965.2814319
[44] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old Is New Again: Binary Security of {WebAssembly}. In *29th USENIX Security Symposium (USENIX Security 20)*. 217–234.
[45] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1045–1058.
[46] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Pecise Types from WebAssembly Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 410–425. https://doi.org/10.1145/3519939.3523449
[47] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly. https://doi.org/10.48550/arXiv.2110.15433 arXiv:2110.15433 [cs]
[48] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 9–20. https://doi.org/10.1145/3460945.3464952
[49] Jan Kasper Martinsen and Håkan Grahn. 2011. A Methodology for Evaluating JavaScript Execution Behavior in Interactive Web Applications. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*. 241–248. https://doi.org/10.1109/AICCSA.2011.6126611
[50] Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. 2011. A Comparative Evaluation of JavaScript Execution Behavior. In *Web Engineering (Lecture Notes in Computer Science)*, Sören Auer, Oscar Díaz, and George A. Papadopoulos (Eds.). Springer, Berlin, Heidelberg, 399–402. https://doi.org/10.1007/978-3-642-22233-7_35
[51] Judy McConnell. [n. d.]. WebAssembly Support Now Shipping in All Major Browsers | The Mozilla Blog. https://blog.mozilla.org/en/mozilla/webassembly-in-browsers/.
[52] MDN contributors. [n. d.]. Performance.Now() - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.
[53] Benedikt Meurer. [n. d.]. An Overview of the TurboFan Compiler.
[54] Mozilla. [n. d.]. Download the Fastest Firefox Ever. https://www.mozilla.org/en-US/firefox/new/.
[55] Paul Muntean, Sebastian Würl, Jens Grossklags, and Claudia Eckert. 2018. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I. 3–25. https://doi.org/10.1007/978-3-319-99073-6_1
[56] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening {WebAssembly} against Spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. 1433–1450.
[57] Node.js. [n. d.]. Node.Js. https://nodejs.org/en/.
[58] Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight Self-Protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09)*. Association for Computing Machinery, New York, NY, USA, 47–60. https://doi.org/10.1145/1533057.1533067

[59] Louis-Noël Pouchet and Tomofumi Yuki. [n. d.]. PolyBench/C. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[60] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1256–1274. https://doi.org/10.1109/SP.2019.00064

[61] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. [n. d.]. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. ([n. d.]), 16.

[62] 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)*. 42–54. https://doi.org/10.1109/ASE51524.2021.9678776

[63] Alan Romano and Weihang Wang. 2020. WASim: Understanding WebAssembly Applications through Classification. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1321–1325. https://doi.org/10.1145/3324884.3415293

[64] Alan Romano and Weihang Wang. 2020. WasmView: Visual Testing for WebAssembly Applications. In *Proceedings of the 42nd International Conference on Software Engineering Companion*.

[65] Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1129–1140. https://doi.org/10.1145/3324884.3416580

[66] Daniel Smilkov, Nikhil Thorat, and Ann Yuan. [n. d.]. Introducing the WebAssembly Backend for TensorFlow.Js.

[67] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

[68] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and Exploiting Optimal Function Inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 977–989. https://doi.org/10.1145/3503222.3507744

[69] Marco Trivellato. [n. d.]. WebAssembly Is Here! https://blog.unity.com/technology/webassembly-is-here.

[70] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 77:1–77:29. https://doi.org/10.1145/3290390

[71] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 133:1–133:28. https://doi.org/10.1145/3360559

[72] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the Performance of Webassembly Applications. In *Proceedings of the 21st ACM Internet Measurement Conference*. ACM, Virtual Event, 533–549. https://doi.org/10.1145/3487552.3487827

[73] Alon Zakai. 2018. The Binaryen Optimizer Goes Up To 4.

[74] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

Received 2023-02-02; accepted 2023-07-27