Exploring Missed Optimizations in WebAssembly Optimizers

Zhibo Liu
The Hong Kong University of Science and Technology
Hong Kong, China
zliudc@cse.ust.hk

Dongwei Xiao
The Hong Kong University of Science and Technology
Hong Kong, China
dxiaoad@cse.ust.hk

Zongjie Li
The Hong Kong University of Science and Technology
Hong Kong, China
zligo@cse.ust.hk

Shuai Wang*
The Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong, China
wei@cse.cuhk.edu.hk

ABSTRACT

The prosperous trend of deploying complex applications to web browsers has boosted the development of WebAssembly (wasm) compilation toolchains. Software written in different high-level programming languages are compiled into wasm executables, which can be executed fast and safely in a virtual machine. The performance of wasm executables depends highly on compiler optimizations. Despite the prosperous use of wasm executables, recent research has indicated that real-world wasm applications are slower than anticipated, suggesting deficiencies in wasm optimizations.

This paper aims to present the first systematic and in-depth understanding of the status quo of wasm optimizations. To do so, we present DITWO, a **di**fferential **t**esting framework to uncover missed optimizations (MO) of **w**asm **o**ptimizers. DITWO compiles a C program into both native x86 executable and wasm executable, and differentiates *optimization indication traces* (OITraces) logged by running each executable to uncover MO. Each OITrace is composed with global variable writes and function calls, two performance indicators that practically and systematically reflect the optimization degree across wasm and native executables. Our analysis of the official wasm optimizer, `wasm-opt`, successfully identifies 1,293 inputs triggering MO of `wasm-opt`. With extensive manual effort, we identify nine root causes for all MO, and we estimate that fixing discovered MO can result in a performance improvement of at least 17.15%. We also summarize four lessons from our findings to deliver better wasm optimizations.

CCS CONCEPTS

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

*Corresponding Author

KEYWORDS

Software Testing, WebAssembly, Compiler Optimization

ACM Reference Format:
Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring Missed Optimizations in WebAssembly Optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598068

1 INTRODUCTION

WebAssembly (wasm) is an increasingly important low-level web language [36, 38] with a multitude of source languages compiled to it [3]. It is widely supported in browsers and used by diverse web applications [4, 62, 73], from "serverless" cloud computing [84], to smart contract platforms [6–8], to sandbox libraries in native applications [12, 61], and even as universal bytecode executed by stand-alone wasm runtimes [11, 13–15].

The wasm community has provided wasm compilers for converting popular high-level languages, including C/C++ [23], Rust [10], and Go [9], to wasm executables. Moreover, Binaryen [31], the official wasm compiler infrastructure library, is provided to facilitate the development of wasm compilers. Binaryen's core component, `wasm-opt` [32], comprises classic compile-time optimizations and wasm-specific optimizations to effectively improve wasm code size and speed, aiming to "*make Binaryen powerful enough to be used as a compiler backend by itself.*" To date, `wasm-opt` has been employed by many industrial-level wasm compilers [2, 9, 10, 23, 24, 43, 69].

Holistically, browser vendors promote wasm with the aim of speeding up web applications [38] and replacing JavaScript (JS), which dominates client-side scripting for decades [64]. With tremendous resources invested in developing the wasm ecosystem, the community generally expects wasm to attain performance comparable to that of native code [33, 38]. However, recent works have shown that wasm programs can be twice as slow as native code [44]. It is also found that wasm may not significantly outperform JS in terms of speed and memory use [87].

Previous studies [44, 87] generally attributed wasm's (counter-intuitive) performance deficiency to the ineffective compile-time (and runtime) optimizations. Nevertheless, a systematic characterization of under-optimized wasm code remains absent, let alone the exploration and classification of the root causes in wasm optimizers. Thus, this paper aims to provide a comprehensive and in-depth investigation of missed optimizations (MO) of wasm optimizers. While this may be partially accomplished by reading the wasm optimizer documents and code, in practice the feasibility is limited by the complexity of the wasm optimizer as well as the nature of program optimizations: optimization opportunities may be subtle and certain optimizations are deemed "missed" only when processing specific code emitted by compiler frontends.

In principle, deciding MO of wasm optimizers would require a "ground truth" (e.g., manually crafting some fully-optimized wasm executables) to compare with, which is challenging to obtain. Inspired by contemporary research on testing C compiler optimizations [80], we instead explore a differential testing setting, by treating native x86 executables fully optimized by modern C compilers as the "reference" to unveil MO. This enables an automated, systematic, and scalable testing of wasm optimizers. Overall, we present DiTwo, a **di**fferential **t**esting framework for **w**asm **o**ptimizers. DiTwo differentiates runtime behaviors of wasm binary code and its native x86 counterparts compiled from the same C code to uncover MO.

The key technical challenge is to select proper "performance indicators" from the wasm runtime logs that are practically feasible to compare and uncover the neglect of various wasm optimization opportunities. To do so, DiTwo launches both wasm and native executables to log two indicators: global variable writes and function calls. These logs form a pair of *optimization indication traces* (OITraces) for cross-comparison. According to our observations, global variable writes and function calls are *resilient* across wasm and x86 executables, i.e., they are roughly close regardless of the differences in two executables. Moreover, such indicators are influenced by an extensive amount of optimization passes in both compilation pipelines. Thus, by differentiating OITraces, we compare the wasm optimizations to mature C compiler optimizations; inconsistencies exposed by cross-comparison indicate missed wasm optimization opportunities.

DiTwo is employed to test `wasm-opt`, the prevailing optimizer maintained by the wasm community and is extensively used by most wasm compilers. Thus, MO found in `wasm-opt` can impede delivering fast and portable wasm applications in various platforms. With 16K randomly generated C programs as test inputs, DiTwo uncovers 1,293 inputs that result in under-optimized wasm programs. With about 140 man-hours, we manually diagnose the root causes behind *all* exposed MO. Moreover, with semi-manual study of five real-world applications, we estimate the lower bound of performance improvement, on average 17.15%, after fixing the MO cases. The results indicate the severity of MO identified by DiTwo. We further summarize four lessons to better optimize wasm code. In sum, this research makes the following contributions:

- This work champions an important yet unaddressed focus on wasm optimizations. We aim to uncover and investigate MO, representing hurdles that may considerably undermine the performance of wasm executables.
- To systematically and practically uncover MO, we design DiTwo, a differential testing-based framework. DiTwo cross-compares a wasm executable and its native x86 counterpart over well-selected performance indicators.
- We extensively tested `wasm-opt`, the core component of the official wasm compiler library. We found 1,293 inputs triggering MO. With extensive manual effort, we identified nine root causes, subsuming *all* uncovered MO cases. All root causes are reported to and confirmed by developers. Our empirical evaluation suggests that fixing such MO can notably speedup wasm code, and we summarize four lessons to better optimize wasm code.

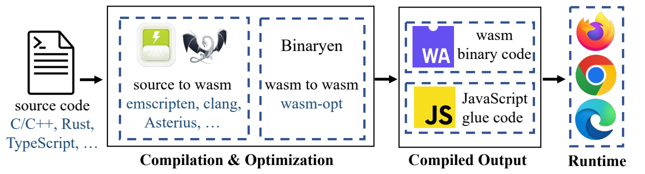


Figure 1: A holistic view of wasm compilation, optimization, and execution pipeline.

Artifact availability. We have released DiTwo [5] for wasm optimizer testing to facilitate further research.

2 BACKGROUND

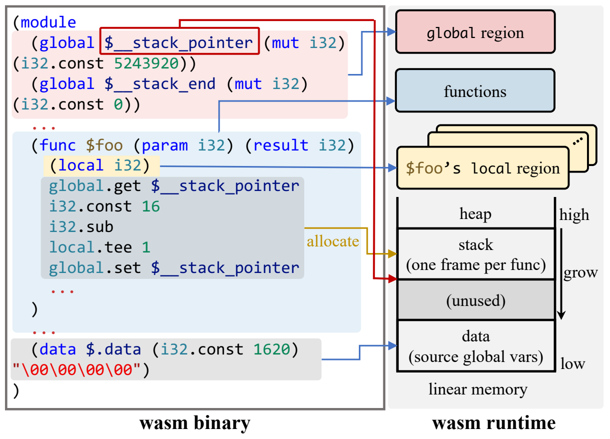
In this paper, we refer to a compiler that compiles high-level language (e.g., C/C++) source code to wasm code as a source-to-wasm (SW) compiler or simply a "wasm compiler." Likewise, we refer to an optimizer that takes wasm code as input, applies multiple optimization passes, and outputs the optimized wasm code, as a wasm-to-wasm (WW) optimizer or simply a "wasm optimizer." We aim to uncover missed wasm optimization opportunities when applying WW optimizer on SW compiler-emitted wasm code.

wasm Compilation. Fig. 1 presents a holistic view of the typical wasm compilation and optimization pipeline. Generally, programs written in high-level languages, such as C/C++, Rust, and Haskell, can be compiled into wasm code and executed in wasm runtimes. In particular, programs written in different languages can be compiled into wasm binary code using existing compiler infrastructures (e.g., with Clang), or the community-offered compilers like Emscripten (`emcc`; for C/C++) and Wasm-Bindgen (for Rust). Then, Binaryen, as the de facto wasm compiler and toolchain infrastructure library, performs various optimizations on wasm binary. Especially, it employs `wasm-opt` to optimize wasm binary code with a set of optimization passes, which significantly impact the code size and runtime performance of the generated wasm binary code.

wasm Execution. wasm application is often not a self-contained wasm executable. wasm code is mainly used to speed up computation-intensive tasks (e.g., 3D rendering), while JS glue code handles invocations of network and IO interfaces. Typically, web applications use JS to instantiate and invoke the interfaces of certain functions, whose underlying implementation is provided in the wasm binary in a high-speed, compact format.

As illustrated in Fig. 1, wasm applications, including the wasm binary and the JS code, are unitedly executed in wasm/JS runtimes, which are often embedded in browsers' JS engines. Unlike JS code which is frequently optimized by Just-in-time (JIT) compilers, runtime optimizations for wasm binary is still an open problem. wasm runtimes leverage techniques varying from ahead-of-time (AOT) compilation to interpretation to execute wasm binary code.

wasm Executable and Runtime Memory Layout. Fig. 2 illustrates a simplified wasm binary (the left half) and its memory layout in the runtime (the right half). Similar to x86 executables, each



```
(module
  (global $__stack_pointer (mut i32)
(i32.const 5243920))
  (global $__stack_end (mut i32)
(i32.const 0))
  ...
  (func $foo (param i32) (result i32)
    (local i32)
    global.get $__stack_pointer
    i32.const 16
    i32.sub
    local.tee 1
    global.set $__stack_pointer
    ...
  )
  ...
  (data $.data (i32.const 1620)
"\00\00\00\00")
)
```

Figure 2: Example of wasm binary and its memory layout.

wasm binary contains multiple sections. First, a wasm binary contains a `global` section (the red region) and a `function` section (the blue region) holding a list of functions. Note that there are no registers in wasm runtime. Instead, intermediate values are stored in local variables (the yellow region). For instance, the `local` memory region for function `foo` is created each time when `foo` is called, and destructed when `foo` returns.

Moreover, the wasm runtime offers a *linear memory region*, denoting a byte-addressable, contiguous memory array. The wasm runtime will not partition the memory section. Rather, SW compilers divide the linear memory into stack frames (for each wasm function call), heap, and global data. Particularly, mimicking the stack in x86 architectures, a memory stack, whose top is pointed by the stack-top pointer (i.e., `$__stack_pointer`; as in Fig. 2), is maintained. The SW compiler-inserted function prefix code allocates a new stack frame each time a function is called.

To clarify, contemporary SW compilers generally insert source code global variables into the "global data region" of the linear memory (at the bottom). In contrast, while the wasm specification (and WW optimizers) anticipates to see global variables in the `global` section (the red region), SW compilers insert only wasm-utility variables into that section, such as the stack-top pointer. Similarly, while the wasm runtime offers a `local` region (the yellow region) for local variables, SW compilers store function local variables in the corresponding stack frame in linear memory. This "mismatch" is reasonable, given that the supported high-level languages like C/C++ were not originally designed for wasm. For instance, wasm does not support pointers, and to mimic pointers in C code, the SW compiler has to put the pointed data *d* into the linear memory, and uses *d*'s offset in the linear memory as its "memory address" for manipulation. This mismatch, however, introduces hurdles for WW optimizers, as uncovered in our evaluation (see Sec. 7).

Tracking Data Access in Linear Memory. Our study focuses on the WW optimization phase where the input, wasm binary code, is parsed into the Binaryen IR, and processed by a rich set of optimization passes in `wasm-opt`. We introduce the research motivation of detecting MO in `wasm-opt` in Sec. 3 and discuss the pipeline of DITWO in Sec. 4.

One performance indicator (i.e., a testing oracle) leveraged by DITWO is variable access patterns. The access patterns of source-level local and global variables can reflect optimization strategies applied over the wasm binary by `wasm-opt`. Compiling wasm binary code with debug information allows each global variable (in the linear memory region) to be easily recognized. Nevertheless, local variable names are removed in wasm binary; it is difficult to track local variables in the linear memory even with the debug information enabled. Thus, we use global variable accesses to form our performance indicator (see Sec. 4) to enable smooth tracking.

3 MOTIVATION

The Demand of wasm-to-wasm (WW) Optimization. With various wasm runtimes and compilers on the market, developing optimization strategies specifically within each SW compiler would be tedious and costly. To avoid reinventing the wheel, the wasm community has advocated optimizing wasm programs at the binary level; this would result in a unified optimizer for different SW compilers. As described in Sec. 2, the official WW optimizer, `wasm-opt`, comprises a rich set of WW optimization passes.

SW compilers may reuse optimizations offered by existing compiler infrastructures, e.g., `emcc` (derived from the LLVM framework) can use LLVM optimization passes prior to generating wasm binary. Nevertheless, we argue that enhancing WW optimizations (this paper's focus) is demanding for two reasons. First, not all SW compilers are accompanied by mature optimization passes, particularly compilers that accept scripting languages (e.g., TypeScript) as input. From the wasm community's perspective, it is unclear when those SW compilers, often maintained by specific programming language communities, would update their optimizers to reach optimization capabilities of mainstream C compilers. Second, existing SW compiler optimizations (such as those in LLVM) are primarily tailored for native executables. Due to the mismatches between the x86 (register-based) execution model and the wasm execution model (i.e., stack-based virtual machine), SW compilers may inevitably generate under-optimized wasm binary [87]. Thus, it is demanding to develop a high-quality WW optimizer that extensively optimizes wasm binary code, the output of SW compilers.

The Status Quo. `wasm-opt` currently provides over 90 optimization passes written in about 26K lines of C++ code. `wasm-opt` employs classic optimization strategies similar to those of mainstream compilers, such as function inlining, dead code elimination, and common subexpression elimination. Besides, `wasm-opt` provides a series of wasm-specific optimizations like memory packing and removing unused branch instructions. `wasm-opt` offers `O0` to `O4` optimization levels, with `Os` and `Oz` focusing on minimizing code size. While these optimizations are compiler- and runtime-independent, `wasm-opt` is expected to complement most compilation-stage optimizations and even works as a standalone compiler backend. It is anticipated that `wasm-opt` can greatly improve code performance before feeding wasm binary to runtimes [1, 2, 32, 43].

Research Objective. Although the concept of "one unified wasm optimizer for all compilers" is appealing, it remains unclear to what extent existing wasm-based optimizations are adequate as a standalone compiler backend. The wasm specification defines a succinct instruction set with around 500 elementary instructions for all arithmetic, memory loading/storing, and control transfers. Unlike native x86 programs, wasm code is deterministic and contains only structured control flows. Therefore, optimizing wasm code would be less complex than optimizing native x86 programs. Moreover, during the WW phase, both wasm-specific and classic compile-time optimizations can be fulfilled, whereas during the SW phase, the optimizations are mostly target-agnostic, potentially losing optimization opportunities and being less comprehensive than the later phase. In practice, however, it is *hard* to implement comprehensive optimization passes at the WW phase. As an assembly-like language, wasm lacks high-level abstraction compared with compiler IR. It becomes technically harder for the optimizer to collect context information from wasm code, as illustrated in *R1–3*; see Sec. 7.1. While recent research has conducted empirical studies to evaluate MO in GCC/Clang [79, 80], they mainly focus on dead code elimination and code size optimization when compiling C code to x86 executables. To systematically assess the status quo of WW optimizations, we detect missed optimizations in live code blocks that are likely executed in runtime. From this point, the issues we found should be more severe, impeding program performance if not properly handled.

We aim to provide an up-to-date assessment of present WW optimizations and identify their MO. We will investigate and categorize root causes of the detected MO cases, and harvest lessons to serve as guidelines for better WW optimizations.

4 APPROACH OVERVIEW

This section describes the challenges in detecting missed optimization opportunities in the WW optimizer, `wasm-opt`. We accordingly present several design considerations that form DITWO, a differential testing-based framework.

Ground Truth. The first challenge to detecting optimization opportunities missed by the WW optimizer is to obtain the optimization *ground truth*, which indicates the optimal optimization state the wasm program can achieve. The ground truth, i.e., fully-optimized wasm executables, is not readily available without extensive manual effort to craft. To shed lights on the "full potential" of WW optimizers, we instead take the modern production C compiler, `Clang`, as the reference to compare with. Specifically, given a C source code P as input, we use `Clang` with the `O3` option to compile it into a fully-optimized native x86 executable E^o_{86}. Next, we compile P into wasm binary E_w with all SW optimizations disabled to leave all optimization opportunities to the WW optimizer. We then config `wasm-opt` with the highest optimization level to optimize E_w into E^o_w. E^o_{86} and E^o_w are compared over several performance indicators (see below) to uncover MO in `wasm-opt`.

Optimization Comparison. Comparing two intensively optimized wasm and x86 binary code, E^o_w and E^o_{86}, to uncover MO in `wasm-opt` is not as simple as expected. We now present and analyze three design considerations below.

Static Comparison. Since wasm follows a stack-based computation paradigm whereas x86 is a register-based computation model, statically comparing instructions in E^o_w with their counterparts in E^o_{86} is challenging: inconsistency in instructions may be due to distinct computation models rather than MO. Moreover, the lack of wasm static analysis tools make it hard to decide if two pieces of wasm instructions and x86 instructions are compiled from the same source code. In conclusion, launching static comparison is hardly feasible.

Coarse-Grained Dynamic Comparison. Given that function names can be retained in E^o_w (when compiling with debug information preserved), we can match each wasm function to its x86 assembly function. Then, to unveil MO in a wasm function, we can execute this function and its x86 counterpart and record their execution time. However, such a function-level, coarse-grained comparison does not provide sufficient information other than that one function is better or worse optimized than another.

Comparing Optimization Indication Traces (OITraces). DITWO extracts two performance indicators, global variable writes and function calls, from the runtime logs of E^o_w and E^o_{86}. These two indicators form a pair of OITraces for comparison, with inconsistencies between the traces indicating MO. Our *key observations* are two-fold: (1) Variable writes are often preceded by a series of computations on the written value. Usually, optimizable computations and the variable write would be optimized away simultaneously, making variable writes a good indicator of how well the code has been optimized. (2) According to recent research [79], function inlining has a substantial impact on program performance, not only by reducing function call costs, but also by increasing opportunities for the rest of the (intra-procedural) optimization pipeline.

Incomplete Debug Information. When compiling C code into x86 binary code, we can enable debug information to annotate functions, global variables, and local variables. Nevertheless, tracking variable writes and function calls imposes a new challenge. Among two well-developed C-to-wasm compilers, `emcc` and Cheerp [78], only `emcc` has mature debugging support while Cheerp cannot attach debug information to wasm binary. However, debug information in wasm binary (inserted by `emcc`) is *incomplete*. In short, the debug information can only help flag each global variable, whereas local variables are hard to track. On the other hand, our test suite, generated by Csmith [88] (see details in Sec. 6), is carefully constructed to involve global variables in most computations. We therefore track and compare only global variable accesses.

Overview. Fig. 3 depicts DITWO's high-level workflow to explore missed opportunities in wasm optimizations. Given a test C program, we compile it into a fully-optimized native x86 executable using `Clang` with `O3` option, and a wasm binary using `emcc` with `O0` option (no optimization). The wasm binary is then optimized using `wasm-opt` with the highest optimization option.[1] Next, the x86 executable and wasm binary are instrumented and executed to log OITraces. Note that our test C programs (generated by Csmith [88]; see Sec. 6) do *not* need user-specific inputs. We use Intel Pin [56] for the x86 executable and implement a static instrumentor for the wasm binary. The logged OITraces are then used for trace consistency checks (inconsistencies denote potential MO cases). Each OITrace comprises global variable writes and function calls, two "performance indicators" that are influenced by an extensive amount of optimization passes in both compilation pipelines. Substantial manual efforts are then spent on studying the root causes

[1]We enable the `-O3` option and extra optimization passes to unleash the full optimization capability of `wasm-opt`; see a list of employed optimization passes at [5].

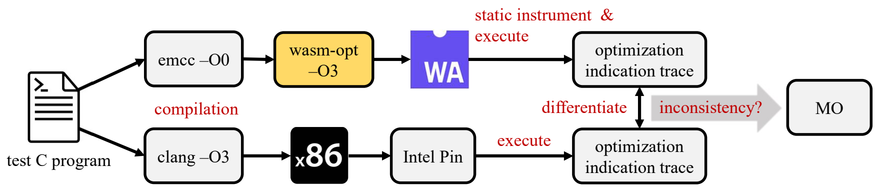


Figure 3: Workflow of detecting MO in wasm optimizers. Our test C programs (generated by Csmith [88]) do *not* need user-specific inputs; see Sec. 6.

of exposed MO. Sec. 5 introduces how we differentiate OITraces in detail.

5 TRACE CONSISTENCY CHECKS

This section introduces two performance indicators, resulting in two types of OITrace consistencies, to detect MO. They are:

Global Variable Write Consistency (GW) checks for redundant assignments (memory writes) to global variables.
Function Call Consistency (FC) checks for function calls that the WW optimizer fails to inline.

Clarification. Note that we do not aim for false negative-free testing, i.e., our trace consistency checks are not complete, and even if both GW and FC are satisfied, potential MO can still exist in wasm-opt-optimized binary. DiTwo uses these two checks, given their generally promising comprehensiveness and high implementation feasibility; as clarified in Sec. 4 and argued by relevant works [79, 80], GW and FC heavily depend on the whole optimization pipeline. Thus, inconsistencies reflect missed GW/FC optimization opportunities, or other MO that have dependency with GW/FC. Overall, we believe through the lens of GW and FC, we shed light on optimizations to quantify how well wasm-opt optimizes wasm binary. Also, other consistency checks could be defined to capture finer-grained program runtime states, and we present relevant discussions in Sec. 8. We now elaborate on checking each indicator below.

5.1 Global Variable Write Consistency (GW)

We assume that if a global variable write is optimized out by the mainstream C compiler, then it should be equally optimizable for wasm-opt. Therefore, we record all global variable writes and check if wasm-opt performs *identical* or *superior* optimizations on global variables than its x86 counterpart. Following notations in Sec. 4, let E^o_{86} and E^o_w be the wasm-opt-optimized wasm executable and its x86 counterpart, respectively, we formulate GW as follows:

Definition 5.1. (GW). For each global variable G encountered during executing E^o_w, we use T^G_w and T^G_{86} to denote two lists of written values toward G when executing E^o_w and E^o_{86}. GW is satisfied if T^G_w is the Longest Common Subsequence (LCS) of T^G_w and T^G_{86}.

Violation of GW. Fig. 4(a) presents a sample code snippet that violates GW when being compiled with emcc and optimized by wasm-opt. In this case, both the global pointer p and the local pointer d point to the same element, $arr[4]$. Thus, the assignment statements at lines 6 and 7 could be merged. However, DiTwo detects that in E^o_w, the written data toward $arr[4]$ is $\{110, 21\}$ (corresponding to two writes at lines 6 and 7), while in its x86 counterpart

```
1.  int32_t arr[9];
2.  int32_t *p = &arr[4];
3.  int8_t foo () {
4.    uint8_t c = 110;
5.    int32_t *d = &arr[4];
6.    *p = c;     //arr[4] = 110
7.    *d ^= 123;  //arr[4] = 21
8.    return *d;
9.  }
10. void main () {
11.   int e = foo();
12.   printf("%d\n", arr[4]+e);
13. }
```
Violation of GW

(a)

```
1.  int32_t a, b, c[] = {8, 8};
2.  int8_t foo (int8_t d) {
3.    for (; d; d += 3)
4.      return d;
5.  }
6.  uint32_t func_1 () {
7.    for (b = 13; b < 27; b++)
8.      a += foo(c[1]) * foo(b);
9.  }
10. int main () {
11.   func_1();
12.   printf("%d, %d\n", a, b);
13. }
```
Violation of FC

(b)

Figure 4: Code examples for GW and FC violations.

E^o_{86}, the written data toward the same variable is $\{21\}$. According to Def. 5.1, we detect a GW violation, given that the wasm trace $\{110, 21\}$ is by no means a LCS of $\{110, 21\}$ and $\{21\}$. With manual inspection, we confirm this finding as an MO case.

Validity of Assumption. One may wonder if register allocation would break this assumption. That is, modern C compilers may store a variable in a register; all writes to that variable will be converted to mov operations to the register and thus will not be tracked by DiTwo. Nonetheless, according to our empirical observation, allocating global variables to registers is rare, possibly due to the complexity of global register allocation in Csmith-generated test cases (see Sec. 6; we use Csmith to generate random test inputs). In fact, after manually investigating all of our findings, we confirmed that no false positives were encountered during our evaluation.

5.2 Function Call Consistency (FC)

To check FC, we record all function calls occurred when executing E^o_w, and compare with those logged when executing the x86 counterpart, E^o_{86}. DiTwo collects the "function call" information by logging each callee function name, all its arguments passed by the caller, and its return value.

With years of improvement, production C compilers can achieve (near) optimal decision in inlining functions for common cases [79]. To lower the cost of function calls and, more importantly, to increase the optimization space for consequent intra-procedural optimizations, wasm-opt should be highly capable (close to the C compiler) of identifying and performing function inlining in wasm code.

Definition 5.2. (FC). For each function F covered during executing E^o_w, we use T^F_w and T^F_{86} to denote two lists of function calls toward F when executing E^o_w and E^o_{86}. FC is satisfied if T^F_w is the LCS of T^F_w and T^F_{86}. Note that to compare two function calls (elements in T^F_w and T^F_{86}), we require that they have the same arguments and return values.

Violation of FC. Fig. 4(b) presents a C code snippet that triggers an FC violation. When compiled with `Clang` and optimized under `O3`, `foo` and `func_1` are both inlined. The inlined function body (loop) allows subsequent intra-procedural analysis to optimize the statement at line 8. Eventually, the optimized x86 executable contains only one `printf` statement with `a` and `b` in constant values. However, `wasm-opt` opts to inline `func_1` while leaving `foo` unchanged. This missed function inlining hinders further optimizations, leaving 28 function calls to `foo` in the wasm binary.

6 IMPLEMENTATION AND STUDY SETUP

DiTwo is implemented in Python, with approximately 3.6K LOC. We use `llvm-dwarfdump` [26] for parsing debug information, and `WABT` [34] for converting wasm binary code into the text format before static instrumentation. In the rest of this section, we report the implementation details of DiTwo and our study setup.

Toolchain. DiTwo compiles C source code into native x86 executables using `clang-12`. wasm currecntly only supports the ILP32-bit platform, i.e., int, long, and pointer are defined as 32 bits. Consequently, all C programs are compiled with the `-m32` option to produce 32-bit x86 executables. We use `emcc` (ver. 3.1.14) to compile C programs into unoptimized wasm binary code (we disable optimization options in `emcc`), and then we use `wasm-opt` provided by Binaryen [31] (ver. 109) to fully optimize wasm binary code.

Instrumenting x86 and wasm Binary. DiTwo compares the OITraces logged during runtime to detect MO. For x86 executables, we use Intel Pin [56], a dynamic binary instrumentor, to (1) hook all function entry points (except C standard library functions) to record function call information (for FC), and (2) instrument each instruction to record its accessed global variable, if any, and the written value (for GW).

An Intel Pin-like dynamic binary instrumentor for wasm does not exist. Thus, we implement static instrumentation by inserting logger code snippets into the wasm binary to record function call and global data access information for FCs and GWs. To avoid possible degradation of wasm optimization due to inserted logger code, we statically instrument `wasm-opt`-optimized wasm code. This way, the inserted logger code is transparent to `wasm-opt`.

Trace Consistency Checks. The trace consistency checks require to compute the LCS, which can be finished in $O(nm)$ time, where n and m are the lengths of two traces. DiTwo maintains a symbol map by parsing the debug information. With the symbol map, a variable's address in wasm binary can be mapped to the same variable's address in the x86 executable, and vice versa. The symbol map is used for pointer value comparison. For instance, given two pointer values in wasm and x86 trace, we deem two pointer values as equal if they both point to the same symbol or both point to unknown symbols. DiTwo currently does not support recursive comparisons of pointers or C structs. If a pointer points to another pointer or a C struct, this pointer is ignored during trace consistency checks.

Testcases. We generate random C test cases with Csmith 2.4.0 [88], while the technical pipeline of DiTwo is independent of the test cases. The C programs generated by Csmith contains complex control flow and a large number of global variables. It does not require user-provided inputs, performs extensive arithmetic computations among global variables, and returns a checksum of all global variables as the output. Randomly generated programs contain plenty of dead code and can be used to stress optimizers. Therefore, Csmith has been used extensively to test C compilers [47, 48, 76]. We use `clang-tidy` [25] to rule out test cases with undefined behavior before usage. We also limit the size of the Csmith-generated code to be less than 50KB to prevent the program from producing excessively huge traces. However, Csmith-generated programs might potentially deviate from real-world programs. To estimate the potential performance improvement (see Sec. 7.2), we also pick five real-world programs from the CHStone benchmark [39] in accordance with previous work [87].

Study Setup. In total, 16,000 C programs were generated for the testing pipeline described in Sec. 4. We ran the experiment on a server with an AMD Ryzen Threadripper 3970X Processor and 256GB RAM. The entire experiment can be finished within 24 hours.

Case Reduction. We use C-Reduce [65] to minimize C programs that trigger MO. Specifically, we assign a unique identifier for each GW/FC. During reducing, we ensure that GWs/FCs related to the MO case are untouched and the MO is still triggered. This step is costly: each time C-Reduce attempts to modify the source code, the testing pipeline depicted in Fig. 3 is invoked to verify if the change is desired. Nonetheless, reducing test cases could largely alleviate the manual efforts required to investigate the root causes of MO.

7 FINDINGS

Overview. We have reported the evaluation setup in Sec. 6. Overall, we use Csmith to generate 16,000 random C programs as test inputs, in which we detected 1,293 programs triggering MO. Two authors then spent approximately 140 man-hours manually investigating the root causes of all exposed MO. Each author has an in-depth knowledge of compiler, binary analysis, and wasm. This ensures the credibility of our findings to a great extent.

We summarize nine root causes of all MO. Links to bug reports are provided on our website [5]. We further harvest four lessons to better optimize wasm code. Besides, we estimate the lower bound of performance gain when the identified MO are fixed. Following, we elaborate on our findings via three research questions. **RQ1**: What are the characteristics of uncovered MO? **RQ2**: What is the potential performance gain after fixing these MO? **RQ3**: What lessons can we deduce from analyzing the MO?

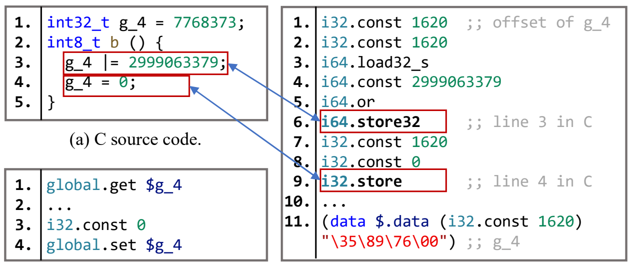
7.1 RQ1: Characteristics of Missed Opt.

We start by answering **RQ1**. To that end, we conduct a labor-intensive manual investigation. Table 1 classifies all the root causes of MO into nine categories. Note that the total number of cases for all root causes exceeds 1,293, as a single case may fall into multiple categories. We elaborate on each root cause below.

R1: Global variables in linear memory As noted in Sec. 2, wasm specification defines the `global` section to store global variables; nevertheless, contemporary SW compilers mainly use this `global` region for wasm-utility variables only, and store source code global variables *implicitly* in the linear memory region. Consider the example in Fig. 5, where g_4 is a global variable in C code. During compilation, the SW compiler uses a fixed memory offset (1620) to represent the variable g_4, and all reads and writes to this variable are converted into `load` and `store` instructions via offset 1620 to the 4-byte memory in the linear memory region.

Table 1: Our testing uncovered 1,293 programs triggering MO. We manually investigate and categorize root causes for all cases in this table.

root causes	R1: global in linear mem	R2: global pointer	R3: local pointer	R4: unopt. loop	R5: func not inlined (stack)	R6: func not inlined (unopt. structure)	R7: func not inlined (lib func call)	R8: insufficient peephole opts	R9: redundant initialization
#cases	108	126	476	767	58	35	8	39	100



```
1. int32_t g_4 = 7768373;
2. int8_t b () {
3.   g_4 |= 2999063379;
4.   g_4 = 0;
5. }
```

(a) C source code.

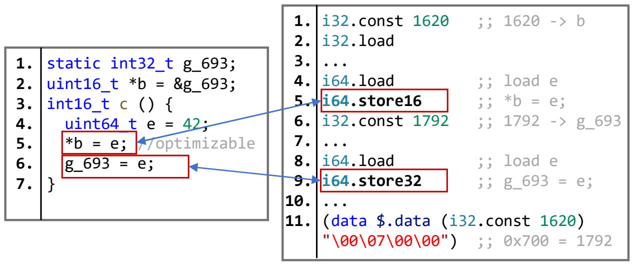
```
1. global.get $g_4
2. ...
3. i32.const 0
4. global.set $g_4
```

(c) Optimizable wasm code pattern.

```
1.  i32.const 1620   ;; offset of g_4
2.  i32.const 1620
3.  i64.load32_s
4.  i64.const 2999063379
5.  i64.or
6.  i64.store32      ;; line 3 in C
7.  i32.const 1620
8.  i32.const 0
9.  i32.store        ;; line 4 in C
10. ...
11. (data $.data (i32.const 1620)
    "\35\89\76\00") ;; g_4
```

(b) `wasm-opt`-optimized wasm code.

Figure 5: Global variable stored in linear memory (R1).



```
1. static int32_t g_693;
2. uint16_t *b = &g_693;
3. int16_t c () {
4.   uint64_t e = 42;
5.   *b = e; //optimizable
6.   g_693 = e;
7. }
```

(a) C source code.

```
1.  i32.const 1620    ;; 1620 -> b
2.  i32.load
3.  ...
4.  i64.load          ;; load e
5.  i64.store16       ;; *b = e;
6.  i32.const 1792    ;; 1792 -> g_693
7.  ...
8.  i64.load          ;; load e
9.  i64.store32       ;; g_693 = e;
10. ...
11. (data $.data (i32.const 1620)
    "\00\07\00\00")   ;; 0x700 = 1792
```

(b) `wasm-opt`-optimized wasm code.

Figure 6: Global variable write via global pointer (R2).

Such code generation patterns notably hinder `wasm-opt` to optimize wasm binary code. The mappings from memory offsets to C global variables are obscure to `wasm-opt`. Without source code or debug information, it is not easy for `wasm-opt` to infer whether two `store` instructions are writing to the same memory region, i.e., statically determining the written addresses of `store` instructions requires flow-sensitive pointer alias analysis, particularly in the case of indirect addressing. In this example, `store` instructions in lines 6 and 9 of Fig. 5(b), corresponding to lines 3 and 4 in Fig. 5(a), both write to *g*_4, and thus the first `store` can be eliminated by optimizations. `wasm-opt` failed to recognize this opportunity.

We find that `wasm-opt` can optimize global variables in the `global` section (the red region in Fig. 2), where each global variable is annotated with a `global` keyword. A case is in Fig. 5(c), where the `global.set/get` instructions explicitly specify accessed variables. `wasm-opt` can correctly optimize out the first global variable write in this case. Overall, the two distinct code styles, although semantically equivalent, could make significant differences for subsequent WW optimizations. We further discuss the possible considerations behind storing global variables in linear memory and alleviations in Sec 7.3.

R2: Writes via global pointers We find 126 MO cases caused by global pointers. As discussed previously, SW compilers are prone to placing C global variables in the linear memory region without their variable name attached. Global pointers are compiled in the same way. Fig. 6 depicts an example. The global pointer *b* is mapped to the memory region beginning at offset 1620, where the hexadecimal offset of *g*_693 (1792) is stored. The `store` instruction at line 5 of Fig. 6(b) writes to the address stored in address 1620, which is read and pushed onto the stack by the `load` instruction at line 2.

The compiled wasm code is functionally correct. Nevertheless, it exacerbates the root cause illustrated in *R1*. With global pointers, it is hardly feasible to infer the written address of each `store` instruction, unless `wasm-opt` is aware of each global variable's address and runtime value during static optimization. As a result, `wasm-opt` fails to decide two `store` instructions access the same global variable, and cannot remove the first `store`.

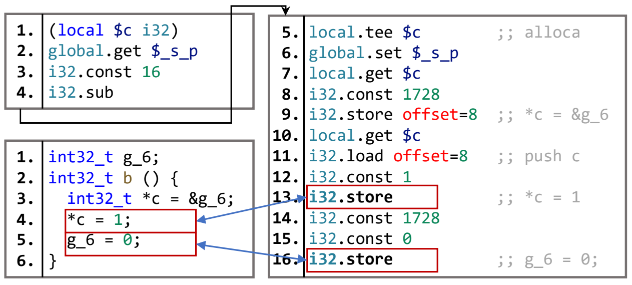
In this case, optimizing the redundant variable write requires `wasm-opt` to determine that the content at address 1620 (line 11 in Fig. 6(b)) denotes a memory address, therefore recognizing a global variable at address 1792. This is a classic challenge in binary code analysis, known as "symbolization" [81, 82], which demands a static analyzer to distinguish memory addresses from constant values. Although many works have attempted to solve this problem, it is not yet resolved in the most general case [27, 28]. Specifically, given that the value stored in address 1620 is 0*x*700, we cannot decide whether it is a constant or an address. However, even though "symbolization" is inherently hard to resolve, it can be circumvented by leveraging debug information that marks pointers. We discuss leveraging debug information in Sec. 7.3. In short, with marked pointers, it is feasible to perform pointer analysis to decide the alias relationship of two `store` instructions and remove the first `store`.

R3: Writes via local pointers We find a total of 476 MO cases due to global variable writes using local pointers. Similar to *R2*, *R3* hinders `wasm-opt` from inferring the written address of `store` instructions, resulting in a more ambiguous mapping between linear memory offsets and global variables.

As shown in Fig. 7(b), at the beginning of the function, the wasm global variable `$__stack_pointer` (abbreviated as `$_s_p`), which points to the top of the memory stack, is subtracted by 16 and updated (lines 2-6). This stack frame allocation operation is analogous to `sub esp, 16` in 32-bit x86 assembly. On line 5, the `local.tee` instruction (assigning the stack address to `c`) allocates stack space for the wasm local variable `c`. At the end of the function, `$__stack_pointer` is incremented by 16 and updated again to deconstruct the stack frame (not shown in Fig. 7 due to space limit).

The ambiguity caused by local pointers is similar to *R2*: the written address of each `store` instruction is unknown, as the optimizer does not know to which variables pointers refer. In contrast to *R2*, the problem caused by local pointers is less complicated. Since the lifetime of local pointers is restricted within a function, we anticipate that `wasm-opt` can statically infer pointer values using mature intra-procedural alias analysis techniques [75].

R4: Loops with global variables In *R1*, we described how SW compilers store global variables in the linear memory region, resulting in a considerable number of MO cases. Such code patterns not only prevent `wasm-opt` from detecting unnecessary global variable writes but also block various follow-up loop-related optimizations,



```
1. (local $c i32)
2. global.get $_s_p
3. i32.const 16
4. i32.sub
```

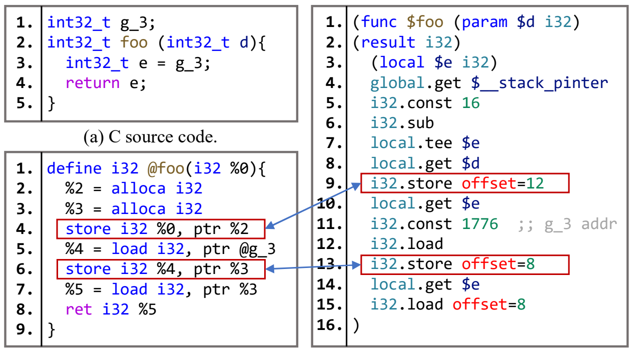
```
1. int32_t g_6;
2. int32_t b () {
3.   int32_t *c = &g_6;
4.   *c = 1;
5.   g_6 = 0;
6. }
```

(a) C source code.

```
5.  local.tee $c          ;; alloca
6.  global.set $_s_p
7.  local.get $c
8.  i32.const 1728
9.  i32.store offset=8    ;; *c = &g_6
10. local.get $c
11. i32.load offset=8     ;; push c
12. i32.const 1
13. i32.store             ;; *c = 1
14. i32.const 1728
15. i32.const 0
16. i32.store             ;; g_6 = 0;
```

(b) wasm-opt-optimized wasm code.

Figure 7: Global variable write via local pointer (R3).



```
1. int32_t g_3;
2. int32_t foo (int32_t d){
3.   int32_t e = g_3;
4.   return e;
5. }
```

(a) C source code.

```
1. define i32 @foo(i32 %0){
2.   %2 = alloca i32
3.   %3 = alloca i32
4.   store i32 %0, ptr %2
5.   %4 = load i32, ptr @g_3
6.   store i32 %4, ptr %3
7.   %5 = load i32, ptr %3
8.   ret i32 %5
9. }
```

(b) LLVM IR code (-O0).

```
1.  (func $foo (param $d i32)
2.  (result i32)
3.    (local $e i32)
4.    global.get $__stack_pinter
5.    i32.const 16
6.    i32.sub
7.    local.tee $e
8.    local.get $d
9.    i32.store offset=12
10.   local.get $e
11.   i32.const 1776  ;; g_3 addr
12.   i32.load
13.   i32.store offset=8
14.   local.get $e
15.   i32.load offset=8
16. )
```

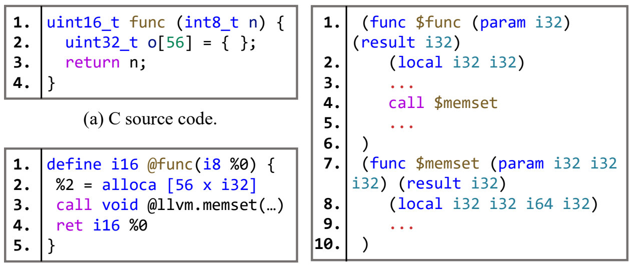
(c) wasm-opt-optimized wasm code.

Figure 8: Redundant store instructions in function (R5).

such as loop invariant code motion, loop reduction, and loop unrolling, especially when loop variables are global. The rational is similar: the target addresses of `load` and `store` instructions are unknown until runtime. Thus, `wasm-opt` cannot statically identify loop variables or loop-invariant code. This observation shows the validity of our approach and the importance of our findings, such that GW consistency check detects under-optimized global variable writes and, more importantly, uncovers subsequent (loop-related) optimization opportunities that are also missed. We find a total of 767 *R4* cases out of 1,293 under-optimized inputs, indicating a considerable optimization space that is absent in `wasm-opt`.

R5 & R6: Stack variables and unoptimized loops in functions As one of the essential compiler optimizations, function inlining replaces function callsites with the callee's function body. Intuitively, function inlining reduces the cost of function calls and, more importantly, brings more opportunities to consequent intra-procedural optimizations to further shrink binary size and improve performance [79]. However, we observed that `wasm-opt` does not always determine the optimal function inlining strategy.

According to our investigation, one primary cause of inline failures is that not-inlined wasm functions contain redundant variables stored in the stack. Considering Fig. 8, where the sample C code is first compiled into LLVM IR (by `emcc` with `O0` optimization), and then into wasm code (the output of `emcc`). Two variables declared at lines 2 and 3 in the LLVM IR are stored in the stack, and accordingly, they are stored on the stack referred to by `$__stack_pointer` in the wasm binary code (Fig. 8(c)). As previously discussed, since wasm `store` instructions can write to any valid address, they are harder to analyze and optimize than LLVM `store` operations, which only write to explicitly pre-allocated memory objects. Consequently,



```
1. uint16_t func (int8_t n) {
2.   uint32_t o[56] = { };
3.   return n;
4. }
```

(a) C source code.

```
1. define i16 @func(i8 %0) {
2.  %2 = alloca [56 x i32]
3.  call void @llvm.memset(…)
4.  ret i16 %0
5. }
```

(b) LLVM IR code (-O0).

```
1.   (func $func (param i32)
   (result i32)
2.     (local i32 i32)
3.     ...
4.     call $memset
5.     ...
6.   )
7.   (func $memset (param i32 i32
   i32) (result i32)
8.     (local i32 i32 i64 i32)
9.     ...
10.  )
```

(c) wasm-opt-optimized wasm code.

Figure 9: Library function in WebAssmebly (R7).

these two wasm `store` instructions (and related code), though redundant, are not optimized. This results in a much lengthy function body and impedes inlining function `$foo`.

Similarly, loops with global variables as loop counters are also hard to optimize (*R4*). Such loops inside wasm functions tend to survive aggressive optimizations, leaving complicated control flows in the optimized functions. Although the conditions that determine whether a function is inlined are complex and varied [20, 70, 89], generally, longer functions are less likely to be inlined. These inlinable functions are thus not chosen for inlining by `wasm-opt`, hindering subsequent optimization passes.

R7: Library function calls in functions C compiler-generated code may implicitly invoke dynamically linked C standard library functions to reduce the binary size or speed up program execution. Nonetheless, when the backend target is set to be wasm and no runtime library is available, such library functions will be compiled into wasm code. Fig. 9 illustrates an example that a function call to `memset` is implicitly inserted when compiling C code into LLVM IR (Fig. 9(b)). In this scenario, LLVM IR-based optimizations can model and optimize the `memset` function call (if we enable optimization levels like `O2`). However, once the LLVM IR is compiled into wasm code (Fig. 9(c)), the library function is supplied with a wasm implementation, whose derived code is not distinguishable from user-coded functions. The wasm implementation of `memset` is not recognized as the standard "memset" utility by `wasm-opt`. Therefore, it requires cumbersome analyses to summarize the `memset` function's semantics. As a result, `wasm-opt` failed to optimize the array allocation (line 2 of Fig. 9(a)) and `func` is not inlined.

R8: Insufficient peephole optimizations As an indispensable component in modern compilers, peephole optimizations (e.g., arithmetic simplifications) are performed by locally rewriting a small set of instructions (known as peephole) without analyzing context information [55, 58]. `wasm-opt` offers a sophisticated peephole optimization pass with a growing list of optimizable instruction patterns [35]. Given that said, we find several missed peephole optimization patterns in the evaluation. The uncovered patterns can be added to the peephole optimization list to augment `wasm-opt`.

We report two examples in Fig. 10. Both examples contain redundant instructions that can be safely removed in the absence of undefined behaviors. In Fig. 10(a), the wasm code snippet is equivalent to $*tmp = *tmp$ in C code, in which the value loaded from address `$tmp` is stored again in the memory region denoted by `$tmp`. All four instructions could be removed if the `store` instruction does not incur undefined behavior (e.g., out-of-boundary). Similarly, in Fig. 10(b), line 3 writes 0 to the address stored in 1620, but another

```
1.  local.tee $tmp
    ;; copy value to tmp
2.  local.get $tmp
    ;; push the value of tmp
3.  i32.load
    ;; read value from *tmp
4.  i32.store
    ;; write value to *tmp

    ;; *tmp = *tmp
```

(a) Pattern 1.

```
1.  i32.const 1620
2.  i32.load
3.  i32.const 0
4.  i32.store   ;;store 0 in *1620
    ...
5.  local.tee $tmp
6.  i64.const 0
7.  i64.store
8.  i32.const 1620
9.  i32.load
10. local.get $tmp
11. i64.load
12. i64.store32 ;;overrides line 4
```

(b) Pattern 2.

Figure 10: Examples of peephole optimization patterns (R8).

```
1. static uint8_t g_359;
2. int main () {
3.   for (g_359 = -26; 0;){
4.     int i = 0, j = 0;
5.     for (; i < 6; i++){
6.       for (; j < 1; j++)
7.         ...
8.   }
9. }
```

(a) C source code.

```
1. i32.const 1728   ;; addr of g_359
2. i32.const 230    ;; 0xE6 (-26)
3. i32.store8
4. ...
```

(b) wasm-opt-optimized wasm code.

```
1. ...
2. (data $.data (i32.const 1728)
3. "\E6")
```

(c) Expected optimized wasm code.

Figure 11: Examples of optimizable initialization (R9).

write later overrides the result at line 10. When both store operations are valid, the first store operation could be eliminated without affecting the final result.

R9: Redundant initialization The last root cause is less significant at first glance but once fixed, it may effectively reduce the binary size and promote follow-up optimizations by creating more optimization opportunities. Fig. 11 depicts a code example, where the global variable g_359 is initialized to −26 at line 3 in the C code. The assignment statement is compiled into wasm code listed in Fig. 11(b), which stores 0xE6 (−26) in the memory block at offset 1728 (address of g_359). The store operation is already in its most succinct form. However, users may expect an alternative, more compact representation: as illustrated in Fig. 11(c), the initialization could be directly moved to the data section in the linear memory.

Once global variable initializations are moved to the data section, wasm binary size can be reduced by removing redundant assignment statements. More importantly, it reduces the number of store instructions that are generally hard to analyze. As discussed in *R1* and *R2*, predicting the target addresses of store and load instructions is generally challenging and frequently hinders wasm-opt. Eliminating such instructions would presumably increase the likelihood of applying subsequent optimizations.

7.2 RQ2: Improvement Estimation

To answer **RQ2**, we aim to evaluate the potential performance improvement gained by fixing MO presented above. However, Csmith-generated programs contain only random computations and control flows, which cannot represent real-world scenarios. Thus, we employ the CHSTone benchmark [39] as per the previous study [87]. Since we are focusing on optimized code, we discard floating-point computations and cryptographic algorithms in the CHStone benchmark, as these programs do not demonstrate a noticeable performance improvement after optimizations. In short, we select all four image/video/audio editing/recognition algorithms and one platform simulation program (MIPS) as our benchmark.

Table 2: Statistics of five real-world programs.

Name	LOC	Description
MIPS	304	Simplified MIPS processor
ADPCM	680	Speech signal processing algorithm
GSM	520	Speech signal processing algorithm
JPEG	2,638	JPEG image decompression
MOTION	709	Motion vector decoding for MPEG-2

Note that we use the modified version of CHStone benchmark provided by a relevant study [87]. The five real-world programs have been edited to be compatible with emcc. Also, since DITWO employs GW, an oracle over global variables, we modify each benchmark program to relocate all local variables as global variables. We clarify that, as in Fig. 2, global and local variables are both in linear memory and are accessed in an *identical* way. Thus, it may be accurate to assume that relocation (from local to global) does not primarily impact the optimization strategies of wasm-opt and its underlying MO issues. Table 2 lists the statistics of our benchmark.

To clarify, fixing all root causes discussed in **RQ1** would require extensive engineering work toward emcc and wasm-opt, which is technically infeasible on our end. Therefore, we are not able to evaluate the ideal (upper bound) performance improvement that could be achieved by fixing all the MO. To evaluate the performance improvement at our best effort, we explore estimating a practical *lower bound* of performance gains after fixing eight out of the nine root causes. The potential improvement of fixing *R4* (optimizable loops) is not estimated as it is difficult to reckon the cost of a redundant loop. We detail our estimation methods below.

Lower Bound. Since patching wasm-opt is technically challenging, we estimate the lower bound of potential performance improvement by assessing the cost of redundant global variable writes (GW) and function calls (FC). Note that this is deemed as a practical "lower bound", given that we omit consequently enabled optimization opportunities when GW/FC are fixed.

It is worth noting that wasm, as a VM specification, does not restrict runtime implementation. The same instruction may incur different costs in different runtimes. Thus, we cannot accurately time the execution of an instruction. Moreover, removing redundant instructions is also difficult. Thus, program execution time before and after fixing is unmeasurable. Instead, we use the number of executed wasm instructions as the program performance indicator.

Lower Bound—GW. To estimate the cost of a redundant GW, we implement a wasm-based *backward* taint analysis to identify instructions related to redundant store instructions. We deem those tainted instructions as optimizable, after the MO are fixed. Experiments on our benchmark indicate that one redundant GW involves an average of 5.35 optimizable instructions. Thus, we estimate the improvement lower bound as the proportion of redundant instructions, i.e., $Lower\ Bound\ (GW) = \frac{5.35 * \#\text{opt. GW}}{\#\text{wasm inst}}$. As shown in the 3rd row of Table 3, the average lower bound (GW) is 13.10%.

One outlier is MOTION, which has only a 0.33% improvement according to our estimation. We discovered that it has limited optimization space w.r.t. GW and FC. Table 4 reports the statistics of MOTION x86 executables. As implied, there is no discernible difference between O0 and O3 binaries in terms of GW, FC, and binary size. Thus, we interpret that this exception as reasonable, and it should not undermine our estimation results in normal circumstances.

Table 3: Performance improvement estimation.

Metrics*	MIPS	ADPCM	GSM	JPEG	MOTION	Average
#wasm inst	82,794	408,453	97,680	8,539,943	82,552	1,842,284
#opt. GW	2,113	22,359	3,919	197,065	51	45,101
Lower Bound (GW)	13.65%	29.29%	21.46%	12.35%	0.33%	13.10%
#opt. FC	0	1,349	475	20,778	3	4,521
Lower Bound (FC)	0%	5.45%	8.02%	4.01%	0.06%	4.05%
Lower Bound (overall)	**13.65%**	**34.74%**	**29.48%**	**16.36%**	**0.39%**	**17.15%**

* "#wasm inst" denotes #executed instructions when executing E^o_w. "#opt. GW" denotes #redundant GWs, "#opt. FC" denotes #inlinable FCs.

Table 4: Statistics of MITION under different compile options.

Metrics	Clang -O0	Clang -O3	GCC -O0	GCC -O3
Binary Size (#assembly inst)	1,301	1,287	1,652	2,054
#FC	32	13	32	11
#GW	8,494	8,441	8,494	8,437

Lower Bound—FC. We dissect the extra cost of a function call into three parts, including passing arguments and return values, stack frame allocation and deconstruction, and storing temporary variables in stack. We use taint analysis to scope these three components to estimate the average function call cost. Our result shows that inlining a function call could save on average 16.5 instructions. We thus estimate the improvement lower bound of inlining function calls as $Lower\ Bound\ (FC) = \frac{16.5 * \text{\#opt. FC}}{\text{\#wasm inst}}$.

The overall (FC+GW) lower bounds are reported in the 6th row in Table 3. On average, fixing all MO discovered in this study would bring approximately a minimum performance improvement of 17.15%. Again, our estimation is conservative, as we do not take the consequent optimization opportunities, after removing redundant GW and performing function inlining, into account. In sum, by exploring and detecting MO to improve wasm-opt with a large margin, the importance of our findings is reasonably justified.

7.3 RQ3: Lessons and Future Improvements

We find that unanticipated code generation patterns applied by SW compilers are one leading cause of MO. Specifically, SW compilers store global variables in the linear memory instead of explicitly declaring them as global variables in wasm. To understand this mismatch, wasm is designed to be safe, fast, and portable. Especially, wasm is a hardware- and platform-independent language with deterministic and easy-to-reason semantics [38]. In contrast, C/C++, despite its efficiency, is notorious for security flaws. Indeed, some flexible C/C++ concepts, such as pointers, are not incorporated in the design of wasm. Thus, SW compilers must employ a workaround to compile C code with pointers to wasm code.

In practice, wasm binary code compiled by emcc stores all global variables in the linear memory and assigns memory addresses to pointers. Local variables are similarly stored in stack frames allocated in linear memory. Such a workaround not only re-introduces security flaws (e.g., buffer overflow) to wasm [49], but also prevents the de facto WW optimizer, wasm-opt, from reaching its full potential. Below, we discuss four lessons harvested from our study.

Minimize the Usage of Stack. As reflected in *R3*, SW compilers are encouraged to allocate local variables into the local region of the wasm runtime (the yellow region in Fig. 2). Local variables should be put in the stack of linear memory only when necessary. To do so, SW compilers should recognize which variables are never referred by pointers. Such variables do not need to be in linear memory to expose their addresses, and can be safely allocated in the local region. Advanced pointer analysis infrastructures [75] can be integrated during compilation to deliver the needed analysis. We deem that pointer analysis simpler in this scenario because the lifetime of a local variable is constrained in a function.

Avoid Storing Global Variables in Linear Memory. Likewise, as reflected in *R1* and *R2*, source-level global variables should be declared in the global region of the wasm runtime (red region in Fig. 2) whenever possible. However, it is more challenging to identify global variables that are never referred to by pointers, requiring sophisticated whole-program pointer analysis. Thus, as an alternative, we advise avoiding using global variables (or not using pointers to point to global variables) when writing wasm applications in high-level languages like C/C++. This way, global variables can be safely stored in the global region for better optimization.

Recover Variables from Memory. As reflected in our study, it is currently unavoidable for wasm-opt to optimize wasm code that exploits linear memory to implement pointers. To bridge the gap between the SW compiler-emitted wasm code and the code expected by wasm-opt, a possible workaround is to convert SW compiler-emitted code into an optimization-friendly form. That is, we need to analyze wasm programs to (partially) recover variables from linear memory (e.g., by extending relevant x86 techniques [16, 17, 41]). The identified variables used in a function can be replaced with the wasm-defined local or global variables.

Employ Debug Information. As discussed in Sec. 7.1, to optimize redundant GWs caused by pointers, wasm-opt needs to identify pointers (and their pointed memory locations) in linear memory to enable follow-up alias analysis. To do so, we advocate for SW compilers to better attach debug information in their emitted wasm code. By leveraging debug information, wasm-opt can easily recognize pointers and variables, thereby greatly easing subsequent WW optimizations. To date, nearly all SW compilers cannot fully attach debug information into their outputs. We deem this as an important improvement to reduce the hurdles of wasm-opt.

8 DISCUSSION

We discuss the validity, extension, and limitation of our technique and findings in this section from the following aspects.

Generalization of Our Findings. While this study extensively detects MO and reveals their hidden root causes, a major threat is that our experiments only cover wasm binary code compiled from C source code. We believe that our findings are not limited in the C-to-wasm context. As discussed in Sec. 7.3, the extensive usage of pointers in C/C++ demands SW compiler to find workarounds when emitting wasm code, which leads to many MO. Note that pointers are not limited to C/C++; hence, we anticipate that "mismatches" between source languages and wasm commonly exist in different SW scenarios and result in many MO. As discovered by recent work [42], over 80% of real-world wasm binaries are compiled from source languages that support pointers like C/C++, Rust, and Go. Moreover, a previous study [49] revealed that the wasm binary compiled from Rust shares a similar linear memory subdivision as the wasm binary compiled from C/C++. We also tentatively explored the wasm binary compiled from Go and found similar code patterns that exploit the linear memory. Therefore, in addition to C/C++, we expect other high-level languages with native x86 executables as compilation targets could encounter similar MO problems when compiled to wasm. We thus believe that our findings are general and instructive for developing other SW compilers.

To clarify, our findings are not specific to Csmith-generated programs. As reflected in Sec 7.2, we found similar MO issues in five real-world programs. Furthermore, our conclusion regarding redundant GW is applicable to local variable writes: for WW optimizers, there is no major difference between local and global variable writes, as both of them are in the linear memory region and accessed in an identical way; thus, we suppose they differ less at the wasm level than that of source level. In sum, while the findings (MO patterns and analyses) are obtained over Csmith programs, our findings and conclusions should be realistic and applicable to real-world wasm code.

Other Languages. The current implementation of DiTwo focuses primarily on emcc, one of the most concerned C-to-wasm compilers in the community. We tentatively explored extending DiTwo to compilers that accept other high-level languages as inputs. However, we noticed that other SW compilers do not show mature support for debug information. For example, Wasm-Bindgen, a Rust-to-wasm compiler, tends to produce incorrect debug information when optimizations are enabled. Therefore, considerably more false positives and negatives will be incurred when using wasm binary code generated by Wasm-Bindgen. Besides, no random program generators comparable to Csmith exist for other high-level languages. Collecting test cases for languages like Rust will require significantly more manual effort. Nevertheless, the differential testing pipeline is mostly *platform-* and *language-independent*. Thus, we envision possible migration of DiTwo to test SW compilers for other languages, when their debug information support is improved.

Other Performance Indicators. The DiTwo prototype logs two performance indicators, global variable writes and function calls, for comparison. While these two indicators are effective at uncovering MO, false negatives are inevitable. Other indicators could be incorporated to suppress false negatives. For instance, arithmetic operations, which reflect a program's computational complexity, may be used to quantify how well a program is optimized. However, due to distinct syntactical forms of x86 and wasm instruction sets, comparing arithmetic consistency necessitates a more elaborate design. We leave exploring other indicators as future work.

9 RELATED WORK

Analysis and Testing of wasm Applications. Wasabi offers the first general-purpose wasm analysis framework for dynamic wasm binary code analysis [50]. Recent works have also explored static analyses like program slicing [74] and taint analysis [29, 77] of wasm binary code. SnowWhite [51] introduced the first learning-based method for recovering high-level parameters and return types of wasm functions. WASim [68] predicts the purpose of a wasm module using machine learning. Given the high demand of fast wasm applications, existing studies have also focused on benchmarking or optimizing the speed of wasm applications [44, 86].

From the security perspective, wasm instruction features a set of design principles (e.g., sandboxes), with the primary goal of protecting host machine from malicious wasm code. However, the wasm code itself is not protected. Recent works have shown that wasm applications suffer from memory exploitations like buffer overflow [49]. A recent empirical study [42] of 8,461 wasm binaries sheds light on the security properties, source languages, and use cases of real-world wasm applications. We also notice several recent works launching fuzz testing and security hardening of wasm applications [30, 46, 57, 59, 71]. Swivel [60] introduced a new compiler framework to protect wasm from Spectre attacks. WAFL proposes a lightweight, VM snapshot-based approach to fuzz wasm binary code [40]. Fuzzm [52] inserts stack canaries and mitigates buffer overflows with static binary rewriting.

Analysis and Testing of wasm Toolchains. We have noticed recent works launching empirical studies to characterize wasm compiler bugs and performance defects [67, 87]. A recent research characterizes standalone wasm runtimes and advocates for effective and low-cost runtime wasm optimizations [83]. In addition to empirical studies, we have also noticed some community efforts launching fuzz testing toward wasm VMs [19, 85]. DiTwo aims to expose and explore more stealthy MO that can induce performance defects. To the best of our knowledge, this has not been comprehensively studied by previous works.

Differential Testing for Systems Software. Differential testing (DT) is used in different software domains, including databases [66, 72], Java Virtual Machines (JVMs) [21, 22], symbolic execution engines [45], disassemblers [63], decompilers [54], and deep learning systems [37]. Beyond finding functionality bugs, recent work also explores locating missed optimization opportunities with DT to improve C compiler infrastructures [18, 80]. CompDiff [53] detects undefined behavior in C/C++ programs with compiler-driven DT.

10 CONCLUSION

This study systematically investigated the hidden MO of wasm optimizers with DiTwo, a differential testing framework to cross-compare wasm executables and their x86 counterparts. Our study exposes a large number of MO. We analyze root causes of all uncovered MO, and outline the key takeaways from this study. This work may serve as a roadmap for researchers and users to improve wasm program performance with optimizers.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedback. The research was supported in part by a RGC ECS grant under the contract 26206520.

REFERENCES

[1] 2021. Binaryen vs LLVM as a compiler backend. https://github.com/WebAssembly/binaryen/discussions/3886.
[2] 2022. AssemblyScript. https://github.com/AssemblyScript/assemblyscript.
[3] 2022. Awesome WebAssembly Languages. https://github.com/appcypher/awesome-wasm-langs.
[4] 2022. Browsers support WebAssembly. https://caniuse.com/?search=WebAssembly.
[5] 2022. Ditwo Artifact. https://github.com/monkbai/wasm-testing.
[6] 2022. Ethereum WebAssembly. https://ewasm.readthedocs.io/en/mkdocs/.
[7] 2022. ink! Documentation. https://use.ink/.
[8] 2022. Terra Wasm Module. https://docs.terra.money/develop/module-specifications/spec-wasm/.
[9] 2022. TinyGo. https://github.com/tinygo-org/tinygo.
[10] 2022. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen.

[11] 2022. wasm3. https://github.com/wasm3/wasm3.

[12] 2022. WasmEdgeRuntime. https://github.com/WasmEdge/WasmEdge.

[13] 2022. wasmer. https://github.com/wasmerio/wasmer.

[14] 2022. wasmi. https://github.com/paritytech/wasmi.

[15] 2022. wasmtime. https://github.com/bytecodealliance/wasmtime.

[16] Gogul Balakrishnan and Thomas Reps. 2010. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.

[17] Gogul Balakrishnan, Thomas Reps, David Melski, and Tim Teitelbaum. 2005. Wysinwyx: What you see is not what you execute. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 202–213.

[18] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th international conference on compiler construction*. 82–92.

[19] bytecodealliance. [n. d.]. Wasmtime Fuzzer. https://github.com/bytecodealliance/wasmtime/tree/main/fuzz.

[20] Dhruva R Chakrabarti and Shin-Ming Liu. 2006. Inline analysis: Beyond selection heuristics. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 12–pp.

[21] 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.

[22] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *PLDI*.

[23] Emscripten Contributors. 2022. Emscripten. https://github.com/emscripten-core/emscripten.

[24] Grain core team. 2022. Grain Compiler. https://github.com/grain-lang/grain.

[25] Clang Developers. 2022. Clang-Tidy - Extra Clang Tools 16.0.0git documentation. https://clang.llvm.org/extra/clang-tidy/.

[26] LLVM Project Developers. 2022. llvm-dwarfdump - dump and verify DWARF debug information. https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html.

[27] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[28] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092.

[29] William Fu, Raymond Lin, and Daniel Inge. 2018. Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint arXiv:1802.01050* (2018).

[30] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*. 123–135.

[31] WebAssembly Group. 2022. Binaryen. https://github.com/WebAssembly/binaryen.

[32] WebAssembly Group. 2022. Binaryen Optimizations. https://github.com/WebAssembly/binaryen#binaryen-optimizations.

[33] WebAssembly Group. 2022. Roadmap - WebAssembly. https://webassembly.org/roadmap/.

[34] WebAssembly Group. 2022. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt.

[35] WebAssembly Group. 2022. wasm-opt peephole optimizations. https://github.com/WebAssembly/binaryen/blob/main/src/passes/OptimizeInstructions.cpp.

[36] WebAssembly Community Group. 2022. Use Cases - WebAssembly. https://webassembly.org/docs/use-cases/.

[37] 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.

[38] 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.

[39] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1192–1195.

[40] Keno Haßler and Dominik Maier. 2021. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-oriented Trends Symposium*. 23–30.

[41] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1667–1680.

[42] 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*. 2696–2708.

[43] Tweag I/O. 2022. Asterius: A Haskell to WebAssembly compiler. https://github.com/tweag/asterius.

[44] 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.

[45] 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.

[46] Matthew Kolosick, Shravan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2021. Isolation Without Taxation: Near Zero Cost Transitions for SFI. *arXiv preprint arXiv:2105.00033* (2021).

[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*.

[48] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*.

[49] 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.

[50] 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.

[51] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 410–425.

[52] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly. *arXiv preprint arXiv:2110.15433* (2021).

[53] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 238–251.

[54] Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487.

[55] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 22–32.

[56] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, 190–200.

[57] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 205–216.

[58] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 448–461.

[59] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. 699–716.

[60] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. 1433–1450.

[61] 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).

[62] Senthil Padmanabhan and Pranav Jha. 2022. WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/.

[63] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version disassembly: differential testing of x86 disassemblers. In *Proceedings of the 19th international symposium on Software testing and analysis*. 265–274.

[64] G Paolini. 1994. Netscape and Sun announce JavaScript, the open cross-platform object scripting language for enterprise networks and the internet. *Press Release]. Sun Microsystems, Inc* (1994).

[65] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[66] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

[67] 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.
[68] Alan Romano and Weihang Wang. 2020. Wasim: Understanding webassembly applications through classification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1321–1325.
[69] The Rust and WebAssembly Working Group. 2022. Shrinking .wasm Code Size - Rust and WebAssembly. https://rustwasm.github.io/docs/book/reference/code-size.html.
[70] Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in line, please! exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. 317–328.
[71] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.
[72] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
[73] Daniel Smilkov, Nikhil Thorat, and Ann Yuan. 2022. Introducing the WebAssembly backend for TensorFlow.js. https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html.
[74] Quentin Stiévenart, David W Binkley, and Coen De Roover. 2022. Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2031–2042.
[75] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.
[76] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*.
[77] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint tracking for webassembly. *arXiv preprint arXiv:1807.08349* (2018).
[78] Leaning Technologies. 2022. An Enterprise-Grade C++ Compiler For The Web. https://leaningtech.com/cheerp/.
[79] 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*. 977–989.
[80] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.
[81] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *NDSS*.
[82] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *USENIX Sec*.
[83] Wenwen Wang. 2022. How Far We've Come–A Characterization Study of Standalone WebAssembly Runtimes. (2022).
[84] wasmCloud Project Authors. 2022. WasmCloud. https://wasmcloud.com/.
[85] wasmerio. [n. d.]. Wasmer Fuzz Testing. https://github.com/wasmerio/wasmer/tree/master/fuzz.
[86] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 1–4.
[87] 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*. 533–549.
[88] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.
[89] Peng Zhao and José Nelson Amaral. 2003. To inline or not to inline? Enhanced inlining decisions. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 405–419.

Received 2023-02-16; accepted 2023-05-03