

That’s a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly

Daniel Lehmann
University of Stuttgart
Stuttgart, Germany
mail@dlehmann.eu

Frank Tip
Northeastern University
Boston, MA, USA
f.tip@northeastern.edu

Michelle Thalakottur
Northeastern University
Boston, MA, USA
thalakottur.m@northeastern.edu

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

WebAssembly is a low-level bytecode format that powers applications and libraries running in browsers, on the server side, and in standalone runtimes. Call graphs are at the core of many inter-procedural static analysis and optimization techniques. However, WebAssembly poses some unique challenges for static call graph construction. Currently, these challenges are neither well understood, nor is it clear to what extent existing techniques address them. This paper presents the first systematic study of WebAssembly-specific challenges for static call graph construction and of the state-of-the-art in call graph analysis. We identify and classify 12 challenges, encode them into a suite of 24 executable microbenchmarks, and measure their prevalence in real-world binaries. These challenges reflect idiosyncrasies of WebAssembly, such as indirect calls via a mutable function table, interactions with the host environment, and unmanaged linear memory. We show that they commonly occur across a set of more than 8,000 real-world binaries. Based on our microbenchmarks and a set of executable real-world binaries, we then study the soundness and precision of four existing static analyses. Our findings include that, surprisingly, all of the existing techniques are unsound, without this being documented anywhere. We envision our work to provide guidance for improving static call graph construction for WebAssembly. In particular, the presented microbenchmarks will enable future work to check whether an analysis supports challenging language features, and to quantify its soundness and precision.

ACM Reference Format:

Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That’s a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA 2023, 17-21 July, 2023, Seattle, USA

© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM... \$15.00
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

WebAssembly [5, 26] is a portable, low-level bytecode format that was introduced in 2017, originally designed for computationally intensive tasks in the browser, e.g., codecs, cryptography, and games. WebAssembly is typically used as a compilation target, i.e., WebAssembly programs are generally not written by hand, but compiled from a variety of source languages, including C, C++, Rust, and Go [28]. Today, WebAssembly is—as envisioned—widely used on the client-side, but also increasingly popular on the server-side, both in Node.js [11] and in standalone runtimes, e.g., Wasmtime [4]. For example, WebAssembly powers real-world websites, such as Adobe Photoshop Web, AutoCAD Web, Google Meet and Keep, Figma, TikTok, Spotify, and WhatsApp Web, widely used browser extensions, such as uBlock Origin, Lastpass, and 1Password, and popular NPM packages, such as OpenCV and TensorFlow.js. Clearly, WebAssembly is already very successful and will remain an important program representation for years to come.

As WebAssembly is gaining in popularity, so is statically analyzing WebAssembly binaries. Widely used industry tools for static analysis and optimization include WABT [14], Binaryen [7], and Twiggy [13]. In academia, Stievenart et al. present a static information flow analysis for detecting security vulnerabilities [46], a static slicing technique for WebAssembly with applications in reverse engineering, code comprehension, and security [44], and the design of a general-purpose static analysis framework for WebAssembly [45].

Call graphs are at the core of many inter-procedural static analyses and are also useful in their own right, e.g., for detecting unused code that can be removed from a binary, called *debloating*, and for understanding and reverse engineering binaries, e.g., when analyzing malware. The tools mentioned above all produce a call graph of a given WebAssembly binary, either explicitly, e.g., depicted visually, or implicitly during their analysis.

Unfortunately, statically computing a sound and precise call graph is challenging. “Sound” here means that the graph contains all call edges that may occur during an execution, and “precise” means that the graph contains only those call edges that may occur during an execution. While computing a sound and precise call graph is impossible for all programs due to Rice’s theorem, there are a number of challenges that are specific to certain languages and program representations. For example, in WebAssembly, indirect calls retrieve the target function by indexing into a table of (function) references, so the precision of computed call graphs crucially

depends on the ability to reason precisely about index values and the state of the table. While challenges of call graph construction have been studied extensively for other languages, e.g., C [33] and Java [40, 47], to the best of our knowledge, no systematic study of the challenges associated with WebAssembly exists. In particular, it is unclear what assumptions are made by existing analyses and how those affect the soundness and precision of the resulting call graphs.

This paper presents the first systematic study of static call graph construction for WebAssembly. We investigate challenges specific to WebAssembly, measure their prevalence, and evaluate the call graphs produced by four existing approaches: Wassail by Stievenart et al. [45], MetaDCE from the Binaryen tool suite [7], the binary code size profiler Twiggy [13], and the call graph of LLVM IR that was lifted from WebAssembly with WAVM [2]. Our study addresses the following research questions:

- **Which WebAssembly language constructs pose challenges for static call graph construction and how commonly are these constructs used in real-world programs?** To answer this question, we codify 12 challenges grouped into six categories, and measure their prevalence on more than 8,000 real-world WebAssembly binaries [28]. (Section 3)
- **What assumptions do current call graph analyses make, and how do these assumptions affect the soundness and precision of the resulting call graphs?** To answer this question, we create a set of 24 microbenchmarks designed to exercise specific soundness and precision challenges. For each microbenchmark, we manually determine the sound and precise ground-truth call graph and compare it with results of the existing analyses mentioned above. (Section 4)
- **To what extent are existing analyses sound for large, real-world binaries?** To answer this question, we dynamically analyze a set of ten large, real-world WebAssembly binaries, compiled from more than 2 million lines of source code, and compare the executed functions to the call graphs and reachable functions determined by the static analyses. (Section 5)

Our study has several noteworthy results. First, we find that WebAssembly-specific challenges are common across real-world binaries. For example, 83% of all binaries use indirect calls through the table section, 64% of all indirect calls are made from function pointers retrieved from (untyped) memory, and three quarters of all binaries lack name or debug information to identify special functions, such as allocators. Second, applying existing analyses to our microbenchmarks reveals that none of the four analyses is sound. In the best case, an analysis is sound for 22 of 24 microbenchmarks, in the worst case for only 7 of 24. Precision leaves even more to be desired, as call graphs are precise for at best 9 of the 24 benchmarks. Third, we find that the soundness problems in existing call graph analyses also affect real-world binaries. Three of four analyses consider functions as unreachable, even though these functions are actually executed in test cases. Several of our microbenchmarks, and even some of the real-world binaries, also lead to crashes in the existing analyses tools. For example, a crash in the Twiggy code size profiler, may prevent developers from size-profiling their binaries, and thus prevent them from reducing binary bloat.

In summary, this paper contributes the following:

```

1 | int square(int x) { return x * x; }
2 | size_t strlen(const char *s) { ... }

```

compiled with Emscripten or Clang to:

```

1 | (module ;; Functions/instructions are statically typed.
2 |   (func (;0;) (param i32) (result i32) ;; square()
3 |     local.get 0 ;; Push the 0th local (the parameter)
4 |     local.get 0 ;; onto the evaluation stack, twice.
5 |     i32.mul ;; Pop 2 arguments, multiply, push result.
6 |   ) ;; Implicit return of the value(s) on the stack.
7 |   (func (;1;) (param i32) (result i32) ... )
8 | ... )

```

Figure 1: Example of C code compiled to WebAssembly.

- The first systematic categorization of WebAssembly-specific challenges of static call graph construction.
- Empirical evidence on how prevalent those challenges are in real-world binaries.
- A set of microbenchmarks and ground-truth call graphs reflecting the challenges, which can be used to assess the soundness and precision of existing and future analyses.
- An evaluation of existing call graph analyses, both on the previously mentioned set of microbenchmarks and on a set of real-world programs, in which we observe several sources of unsoundness and imprecision.

By enumerating the specific challenges that must be overcome by call graph analyses, and providing a set of microbenchmarks and real-world programs that can be used to assess soundness and precision, our results will help future call graph analyses to avoid known pitfalls. We make our microbenchmarks, and all code and results publicly available at:

<https://github.com/sola-st/wasm-call-graphs>.

2 BACKGROUND ON WEBASSEMBLY

We give a short introduction to the WebAssembly language and ecosystem. For more details, please refer to the website [5], the initial academic publication [26], and the official language specification [3]. WebAssembly is a compact binary representation that was originally designed for computationally intensive tasks in client-side web applications. It is in widespread use for various tasks, such as image processing, games, or programming language implementations [28]. WebAssembly achieves portability by being hardware-, platform-, and language-independent. It is increasingly used beyond client-side web applications, e.g., on the server-side with Node.js [11] and on standalone runtimes, e.g., Wasmtime [4]. In this case, WASI [1] provides a standardized syscall interface, including file system or network access, akin to POSIX for native code. WebAssembly serves as a compilation target for higher-level languages, e.g., with popular compilers from C/C++ [8, 10], Rust [12], and Go [9]. Unless otherwise specified, this paper refers to WebAssembly 1.0, which is the core language without optional extensions, and supported by all major browsers and standalone runtimes.

Figure 1 shows a simplified WebAssembly program compiled from C. A WebAssembly *module* contains a set of *functions*, where each function takes a list of values as parameters and produces a list

of results. The body of a function consists of a sequence of *instructions*. WebAssembly is a stack-based language, where instructions pop arguments values from an implicit *evaluation stack* and push computed results back onto this stack. Instructions may also refer to *local variables*, *global variables*, and they may load values from and store values into *linear memory*, which is a contiguous, mutable array of raw bytes. A WebAssembly module may also have a function table, or short *table*. Indirect function calls, e.g., originating from dynamically dispatched method calls in a high-level language, are accommodated using a `call_indirect` instruction that takes an index into the function table as an argument.

WebAssembly is a statically typed language, but supports only four low-level types: 32 and 64-bit integers (`i32/i64`), and 32 and 64-bit floating point values (`f32/f64`). Any other types present in higher-level languages, e.g., objects, arrays, pointers and references, must be expressed in terms of these four basic types, in combination with table lookups and accesses to linear memory.

Each WebAssembly module has associated *imports* and *exports*. Each import is a pair (m, e) that is required for instantiating the module. Here, m is the name of another module and e is the name of an entity (function, table, memory, or global) within m . A module's exports define entities that become accessible from the outside once the module has been instantiated. The *instantiation* of a module checks that the provided imports match the declared types and performs initializations. After instantiation, all exported functions from a module can be invoked.

Given the original browser use case, WebAssembly code is designed to interoperate with a *host environment*. To this end, WebAssembly's import mechanism does dual duty as a foreign function interface by allowing the import of *host functions*. In the browser, those are JavaScript functions, and in standalone WASI runtimes, they may be implemented in native code.

3 CHALLENGES AND THEIR PREVALENCE

This section presents our classification of challenges for static call graph construction in WebAssembly (Section 3.1) and their prevalence in real-world binaries in Section 3.2. In total, we identify 12 WebAssembly-specific challenges, as summarized in Figure 2.

We identified and classified the challenges based on three initial investigations. First, we started with manual inspection of real-world binaries. Second, we came across challenges while running existing analysis tools and inspecting the analyses' source code to see why they crashed or were unsound. Finally, we systematically went through the official WebAssembly specification and checked for each type of section in the binary and each kind of instruction whether it could have an effect on call graph analysis. For classifying the collected challenges into categories, we iterated and discussed among the authors. While this gives no formal guarantee that no further challenges (e.g., specific to certain source programming languages) will be uncovered in the future, we are confident that all sections and instructions of WebAssembly are considered. In particular, there are only two call instructions in WebAssembly 1.0 (`call` and `call_indirect`), which we both discuss, and our challenges touch on each binary section kind in the official specification.¹

Program representation:

- CFunctionIndices** No function names, only indices.
- CProgramStructure** Low-level program structure, static linkage.

Indirect calls and table section:

- CTableIndirection** Double indirection in indirect calls requires knowledge about table contents.
- CTableIndexValue** Table index values (function pointers) are hard to identify and determine statically.
- CTableInitialState** Table initialization may depend on host code.
- CTableMutation** Table may be mutated by host code.

Types:

- CLowLevelTypes** Types are low-level and imprecise.

Host environment:

- CHostCallbacks** Outgoing edges of imported functions.
- CEntryPoints** No standard entry point(s).

Memory:

- CMemoryMgmt** No built-in memory management.
- CMemoryMutable** Linear memory is writable everywhere.

Source languages:

- CMultiPL** Binaries are compiled from different source languages, there is no standard library.

Figure 2: Overview of the WebAssembly-specific challenges.

When identifying challenges unique to WebAssembly, we also compare the language against other program representations, such as source code of high-level languages (e.g., Java), native machine code, and against other bytecode formats (e.g., JVM bytecode).

3.1 Challenges

3.1.1 Program Representation. Unlike in source code or JVM bytecode, functions in WebAssembly are not identified by names but with integer indices (**CFunctionIndices**). For example, a direct call to the twenty-fourth function is just `call 23` in the binary. The index does not convey any semantics or intuition about the function, making it difficult to identify special-purpose functions, e.g., `memcpy`. Binaries may contain debug information, but as we will show, few in the wild do.

A related challenge is WebAssembly's low-level program structure (**CProgramStructure**). All functions are laid out in a single flat index space, without any structure to express source-level concepts such as scopes or classes. Unlike in source code or JVM bytecode, it is not known, for example, whether a function is a method, and of which class. WebAssembly code is also statically linked, such that functions from different libraries (including, e.g., the C standard library) cannot be easily distinguished. Consider the binary in Figure 4a as an example. It has four functions, but no nesting or higher-level organization, and no source language concepts, such as scopes, classes, or namespaces.

3.1.2 Indirect Calls and Table Section. Besides regular direct calls, WebAssembly has *indirect calls*, where the call target is determined at runtime. Those implement function pointers, virtual calls, and

¹<https://www.w3.org/TR/wasm-core-1/#modules%E2%91%A0>

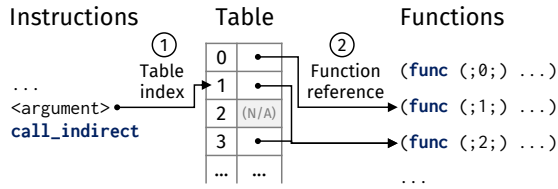


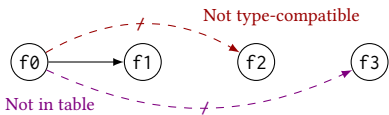
Figure 3: Illustrating the double indirection via the table.

```

1 (module
2   (func (;0;)
3     instruction sequence producing an i32 index...
4     ;; Calls the function at the given table index;
5     ;; the function must have type [] -> [i32].
6     call_indirect (type (param) (result i32))
7     ... ;; Rest of the function...
8   )
9   ;; Other functions in the binary:
10  (func (;1;) (param) (result i32) ... )
11  (func (;2;) (param i32) (result i32) ... )
12  (func (;3;) (param) (result i32) ... )
13  ;; Table section: contains function references,
14  (table funcref)
15  ;; initialized with 2 entries starting at offset 0.
16  (elem (i32.const 0) func 1 2) ;; f3 is not in table.
17 )

```

(a) WebAssembly bytecode.



(b) Simplified call graph. The dashed edges can be removed because f2 is not type-compatible with the call instruction in f0 and because f3 is not present in the table (assuming the table is immutable).

Figure 4: Example of indirect calls in WebAssembly.

other forms of dynamic dispatch. WebAssembly 1.0 has only a single instruction for this purpose, `call_indirect`. As shown in the example of Figure 4, the `call_indirect` instruction pops an `i32` value from the evaluation stack, uses this value to retrieve a function reference from the `table` section, and then invokes the function. This lookup scheme introduces a *double indirection* ($C_{TableIndirection}$), as illustrated in Figure 3. To determine an indirect call’s potential targets, an analysis thus needs to reason about two things: First, the index value, and second, the table contents.

Determining the index value of an indirect call is a challenge in its own right ($C_{TableIndexValue}$) since index values are of type `i32`, i.e., indistinguishable from other pointers or regular numbers, and they can be manipulated like any integer. This differs from higher-level representations, where function pointers can be identified by type, and even from native code, where code pointers can be identified because they point into a code memory range.

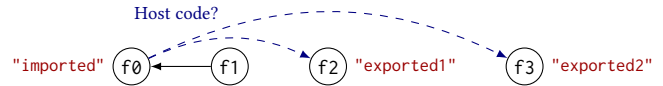
We identify two further challenges specific to determining the state of the table. The first is about the initial state of the table ($C_{TableInitialState}$). A table is initialized via the `elem` section (line 16

```

1 (module
2   (import "host" "imported" (func (;0;)))
3   (func (;1;)
4     call 0 ;; Does func 0 call exported function(s)?
5   )
6   (func (;2;) (export "exported1") ... )
7   (func (;3;) (export "exported2") ... )
8 )

```

(a) WebAssembly bytecode.



(b) Simplified call graph for (a). The presence of the blue dashed edges depends on (assumptions about) the host code of f0.

Figure 5: Example of imported and exported functions.

of Figure 4a), which copies a list of function references at a specific offset into the table during module instantiation. The offset itself may be an imported global variable, hence an analysis would require to analyze also the host code (which may be in JavaScript or even native code). Alternatively, the whole table itself may be imported from the host, posing the same difficulty. To make matters worse, tables may be mutated at any point in time ($C_{TableMutation}$). While this is not possible within WebAssembly 1.0 itself, the host environment can do so, e.g., in browsers using the JavaScript function `WebAssembly.Table.set()`.

3.1.3 Types. WebAssembly’s four low-level types are of limited help for constructing call graphs ($C_{LowLevelTypes}$). On the upside, the `call_indirect` instruction statically encodes the type of the target function (line 6 in Figure 4a), which constrains the set of call targets at least somewhat (Figure 4b). However, many source-level types are compiled to the same low-level WebAssembly type. Figure 1 illustrates that all three C types `size_t`, `const char *`, and `int` get compiled to the same WebAssembly type `i32`. Thus, even though indirect call targets are restricted by type, many functions are type-compatible with each other. The absence of richer types implies that type-based call graph algorithms, e.g., RTA or XTA [19, 49] as used for Java, are not applicable to WebAssembly.

3.1.4 Host Environment. WebAssembly binaries execute within a host environment (e.g., the JavaScript environment of a browser), so each WebAssembly binary is only a partial program. Determining its *entry points* is one challenge ($C_{EntryPoints}$). Some binaries have designated *start* functions, but other functions may be entry points as well. Another challenge is that functions imported into WebAssembly may call any function reachable from the host ($C_{HostCallbacks}$). E.g., in Figure 5, function `f0` could be a host function that is implemented in JavaScript and which calls either of `f2` or `f3`. By analyzing the binary alone, an analysis has no way of precisely determining whether such calls are possible.

3.1.5 Memory. Two challenges are associated with WebAssembly’s notion of *linear memory*. First, WebAssembly does not provide any garbage collection or other high-level memory management ($C_{MemoryMgmt}$), so memory must be managed by functions compiled into the binary itself. As a result, `malloc` and `free` are just

Affected binaries		Property	Challenge(s)
Percentage	Nb.		
1	74%	6,239 Has no function name section	CFunctionIndices
2	95%	8,014 Has no .debug section	CFunctionIndices
3	83%	6,961 At least one indirect call	CTableIndirection
4	22%	1,846 Table is imported or exported	CTableInitialState CTableMutation CHostCallbacks
5	12%	1,084 Element offset from variable	CTableInitialState
6	92%	7,680 At least one imported function	CHostCallbacks
7	97%	8,144 No explicit start function	CEntryPoints
8	87%	7,339 No explicit WASI start function	CEntryPoints
9	95%	7,993 At least one store instruction	CMemoryMutable

Figure 6: Measurements on binaries in the WasmBench dataset, and to which challenges they relate.

regular functions in the binary, and call graph algorithms that reason about allocation sites or types of objects [19] are not easily applicable. Second, WebAssembly differs from native code in that *all linear memory is writable everywhere* (**CMemoryMutable**); even data that typically do not change during execution, e.g., vtables and static data, are potentially mutable.

3.1.6 Source Languages. WebAssembly binaries are compiled from a variety of different source languages, including C, C++, Rust, Go, and AssemblyScript [28]. Analyses that make assumptions about the source language, e.g., the presence of vtables in binaries compiled from C++, are therefore only applicable to a subset of all binaries (**CMultiPL**). On a related note, there is no single standard library, for which one could special-case call graph construction.

Insight. WebAssembly’s idiosyncratic features (e.g., indirect calls, tables, low-level types, and mutable linear memory) pose many challenges for call graph analysis. These challenges differ from high-level source code, native machine code, and other bytecode.

Implication. The unique challenges motivate work on future call graph analyses tailored specifically to WebAssembly.

3.2 Prevalence of Challenges in the Wild

To assess whether the identified challenges appear in practice, we measure their prevalence on a large set of real-world binaries.

3.2.1 Dataset and Setup. We base our analysis on the WasmBench dataset [28], which contains 8,461 binaries collected from websites, browser extensions, NPM, and GitHub repositories. Our study considers all 8,392 binaries that can be successfully parsed as WebAssembly 1.0 without any extensions. We statically analyze this dataset to quantify how many binaries or functions are potentially affected by the identified challenges, except for challenges that are general properties of the language, **CProgramStructure** and **CMemoryMgmt**, and thus always applicable.

3.2.2 Results. Figures 6, 7, and 8 depict the results. Figure 6 shows how many binaries are affected by certain properties and which challenges they relate to. From the first two rows, we see that

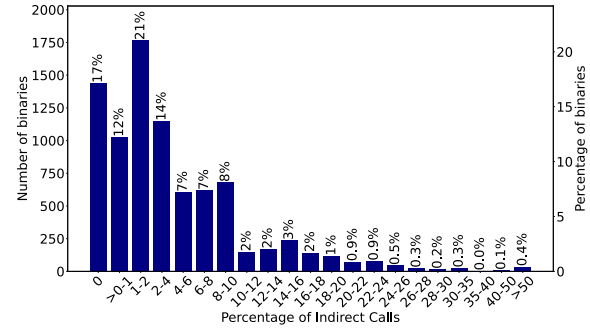


Figure 7: Distribution of indirect calls in the binaries.

CFunctionIndices applies to most binaries in the wild. 74% of all binaries have no name section and thus lack names for their functions. Full debug information is absent from 95% of the binaries. Hence, in most binaries, no textual or source information is associated with functions that could help identify them.

Rows 3 to 5 are related to indirect calls and the function table. First, we see that 83% of all binaries contain at least one indirect call, and hence are affected by **CTableIndirection**. We also see that 22% of all binaries have either an imported or exported table. In such cases, the host either initializes the table (**CTableInitialState**), or can mutate it during runtime (**CTableMutation**), so the state of the table cannot be determined from the binary alone. Moreover, the fact that all functions in an imported or exported table are reachable from the host may necessitate call graph edges that originate from imported functions (**CHostCallbacks**). In further 12% of binaries, the table itself is not imported, but it is initialized with an element section whose offset is an imported variable, again necessitating analysis of the host code.

Rows 6 to 8 relate to entry points and host callbacks. We see that 92% of all binaries import at least one function (not shown: on average, 18% of all functions are imported), indicating that analyzing or modeling host code is crucial. In terms of entry points, we see that only a small fraction of binaries specify an explicit entry point, either via the WebAssembly `start` section (3%) or via WASI’s `_start` function (13%). Selecting sound and precise entry points for an inter-procedural analysis thus remains a challenge for more than 84% of the binaries. Not shown but also related to **CHostCallbacks** is that, on average, 22% functions are exported and 16% are present in an element section. All such functions are reachable from the host and could potentially be called from imported functions.

Rows 7 and 8 of Figure 6 are concerned with linear memory. Almost all (95%) of all binaries contain at least one store instruction and make use of memory, requiring pointer analysis for precision and suffering from WebAssembly’s always writable memory.

Finally, we refer to Hilbig et al. [28] for their analysis of the source languages of WebAssembly binaries (**CMultiPL**). They find that there is a wide range of source languages, with 57% of binaries compiled from C/C++, 14% from Rust, 3% from AssemblyScript, 2% from Go, and a long tail of other languages. E.g., if one were to handle only C and C++ binaries, more than 40% of the binaries would remain unsupported.









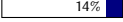


Distribution	Count	Description
	9.6k	Constant table index (<code>i32.const n</code>)
<i>Data-flow related:</i> Index instructions involve...		
	2.1M	local variable (<code>local.get/tee l</code>)
	1.9M	function parameter (<code>local.get/tee p</code>)
	16k	function result (<code>call f/call_indirect</code>)
<i>Memory-related:</i> Index instructions involve...		
	2.7M	at least one load, whose address...
	96k	is a constant (<code>i32.const n</code>)
	364k	involves local variable (<code>local.get/tee l</code>)
	533k	involves parameter (<code>local.get/tee p</code>)
	586k	involves another load (<code>i32.load</code>)
	2.4M	involves some (pointer) arithmetic
<i>Other:</i>		
	3.5M	Index involves some arithmetic (see text)

Figure 8: Breakdown of $C_{\text{TableIndexValue}}$ by instructions.

Insight. The identified challenges impact many (between 12% and 97%) of the 8,392 binaries under consideration.

We dive more deeply into indirect calls in Figures 7 and 8, as statically approximating runtime dispatch poses a major challenge to call graph analysis. Figure 7 shows a histogram of the fraction of indirect calls (in terms of all calls) over all binaries. On average, indirect calls constitute 4.9% (mean) and 2% (median) of all calls. In total, the dataset contains 4.2 million indirect calls.

At this point, the reader may wonder if some indirect calls may be handled specially. In Figure 8, we analyze the sequences of instructions preceding `call_indirect`s in the dataset. Those instructions determine the index value ($C_{\text{TableIndexValue}}$) and hence (together with the table) which function(s) may be called. The first row makes clear that more than 99% of index values are *not* constant and thus not easily determined. Instead, the next three rows show that many index values require some form of data-flow analysis. In WebAssembly, the `local.get` and `local.tee` instructions do double duty to refer to both function parameters and local variables. We take the type signature of the current function into account to distinguish between references to parameters and local variables. The first row shows that tracking data-flow through local variables is crucial, as 49% of index values flow out from an access of a local variable. 45% of index values require even inter-procedural data-flow analysis, since they are read from the current function parameters or, to a much lesser degree, the return value of another function.

In many cases, the index into the function table for indirect calls is itself retrieved from memory first. The next six rows thus analyze the kinds of memory accesses in the instructions that compute the table index value. 64% of the instruction sequences that compute an index value contain a memory load and hence require some form of pointer analysis. We distinguish between different cases of increasing difficulty. In the simplest case, the address of the load is itself a constant (e.g., globally allocated function pointers). This is however only the case for 2% of all indirect calls. Instead, in many cases the address involves a local variable (9%), a function parameter (13%), or another load (14%). Double indirect loads are

common, e.g., for C++ virtual method calls, where the first load retrieves the `vtable` of an object and the second the function pointer inside that `vtable`. In 58% of all indirect calls, the address given to a load also involves some arithmetic operation, likely for pointer arithmetic, e.g., when accessing array elements or struct fields.

If one considers arithmetic operations in all instruction sequences for table index values, they appear 86% of the time. The most common arithmetic operations in table index expressions are `i32.add` (20% of all instructions), `i32.and` (11%), `i32.shl/i32.shr_u` (0.5%), and `i32.sub` (0.1%). Clearly, addition and bitmasking account for the most important arithmetic operations.

Insight. Index values for indirect calls are frequently not constant, but instead involve intra- and inter-procedural data-flow, memory loads, and arithmetic operations, such as addition and bitmasking.

Implication. To ensure precision, a static call graph analysis for WebAssembly should track data-flow and perform pointer analysis.

4 EVALUATION ON MICROBENCHMARKS

Given the unique challenges in WebAssembly, how do existing static analyses compute call graphs, and how sound and precise are they? To answer that, we introduce a set of microbenchmarks (Section 4.1) specifically crafted to expose the challenges, and then evaluate four existing call graph analyses (Section 4.2) against them. Every challenge is covered by at least one microbenchmark, but one challenge may be exposed by multiple benchmarks. E.g., $C_{\text{TableIndirection}}$ applies to all programs with at least one indirect call. The results are summarized in Table 1 and detailed in Section 4.3. We make our microbenchmarks, results, and automated scripts to evaluate future analyses on them publicly available in the supplemental material.

4.1 Microbenchmarks

Table 1 gives an overview of our 24 microbenchmarks, with a short description for each and which challenges it relates to. Each microbenchmark has four components:

1. The *WebAssembly binary*. All but one are written directly in the bytecode text format. To be useful for debugging, the binaries are small and aim to test isolated challenges.
2. The *set of entry points*, i.e., functions that are assumed to be reachable and initially called. Benchmarks 4 and 5 give no explicit entry points, to test whether the analysis is aware of language-intrinsic entry points. Analyses for which no entry points can be given should conservatively regard all functions that are reachable from the host as entry points.
3. The ground-truth *call graph*, which we construct manually from the program for all possible inputs. It is a directed, possibly cyclic graph (F_{all}, E) , where F_{all} is the set of all functions and E is the set of edges between them.
4. The *set of reachable functions* F_r , i.e., which functions could potentially execute given any possible program input. We also construct this manually and confirm the functions are reachable by running the program on appropriate inputs.

Components 1 and 2 are inputs to a call graph analysis, whereas components 3 and 4 serve as a ground truth we compare the analyses against. Some analyses only report a call graph, not a set of reachable functions. For those analyses, we obtain the reachable

Table 1: Overview of the microbenchmarks and results of different call graph analyses on them. $|F_{\text{all}}|$ is the total number of functions in the binary. $|F_r|$ is the number of functions reachable from the given entry point(s). $|E|$ is the number of edges in the call graph. S and P indicate that an analysis is sound and precise, respectively, compared to the ground truth.

#	Description	Challenges	Ground Truth			Wassail				WAVM+LLVM				MetaDCE			Twiggy		
			$ F_{\text{all}} $	$ F_r $	$ E $	$ F_r $	$ E $	S	P	$ F_r $	$ E $	S	P	$ F_r $	S	P	$ F_r $	S	P
1	Simple direct call		3	2	1	2	1	✓	✓	2	1	✓	✓	2	✓	✓	2	✓	✓
2	Transitive direct call		5	3	3	3	3	✓	✓	3	3	✓	✓	3	✓	✓	3	✓	✓
3	Direct call to imported function		3	2	1	2	1	✓	✓	2	1	✓	✓	2	✓	✓	2	✓	✓
4	Implicit entry point: Wasm start section	$C_{\text{EntryPoints}}$	2	0	1	0	1	✓	✓	0	1	✓	✓	0	✓	✓	Crash		
5	Implicit entry point: WASI start function	$C_{\text{EntryPoints}}$	3	2	1	2	1	✓	✓	2	1	✓	✓	0	✗	✗	2	✓	✓
6	Imported host code calls exported function	$C_{\text{HostCallbacks}}$	7	6	3	6	3	✓	✓	6	3	✓	✓	2	✗	✗	6	✓	✓
7	Functions in exported table are reachable	$C_{\text{HostCallbacks}}, C_{\text{TableIndirection}}$	5	3	3	2	2	✗	✗	2	2	✗	✗	1	✗	✗	3	✓	✓
8	Functions in imported table are reachable	$C_{\text{HostCallbacks}}, C_{\text{TableIndirection}}$	5	3	3	2	2	✗	✗	2	2	✗	✗	3	✓	✓	2	✗	✗
9	Table is mutated by host	$C_{\text{TableMutation}}, C_{\text{TableIndirection}}$	4	3	1	3	1	✓	✗	3	0	✓	✗	1	✗	✗	3	✓	✓
10	Table init. offset is imported from host	$C_{\text{TableInitialState}}, C_{\text{TableIndirection}}$	3	2	1	Crash				1	0	✗	✗	Crash			3	✓	✗
11	Memory init. offset is imported from host	$C_{\text{TableIndirection}}$	3	2	1	3	2	✓	✗	1	0	✗	✗	Crash			3	✓	✗
12	Functions must be in table for indirect call	$C_{\text{TableIndirection}}$	3	2	1	2	1	✓	✓	1	0	✗	✗	2	✓	✓	2	✓	✓
13	Types can constrain indirect call targets	$C_{\text{LowLevelTypes}}, C_{\text{TableIndirection}}$	3	2	1	2	1	✓	✓	1	0	✗	✗	3	✓	✗	3	✓	✗
14	Constant table index value	$C_{\text{TableIndexValue}}, C_{\text{TableIndirection}}$	3	2	1	3	2	✓	✗	1	0	✗	✗	3	✓	✗	3	✓	✗
15	Index value data-flow through local variable	<i>as above</i>	3	2	1	3	2	✓	✗	1	0	✗	✗	3	✓	✗	3	✓	✗
16	Masked index value	<i>as above</i>	4	3	2	4	3	✓	✗	2	1	✗	✗	4	✓	✗	4	✓	✗
17	Inter-procedural index value, parameter	<i>as above</i>	4	3	2	4	3	✓	✗	2	1	✗	✗	4	✓	✗	4	✓	✗
18	Inter-procedural index value, function result	<i>as above</i>	4	3	2	4	3	✓	✗	2	1	✗	✗	4	✓	✗	4	✓	✗
19	Index from memory, constant address	<i>as above</i>	3	2	1	3	2	✓	✗	1	0	✗	✗	3	✓	✗	3	✓	✗
20	Index from memory, address inter-procedural, parameter	<i>as above</i>	4	3	2	4	3	✓	✗	2	1	✗	✗	4	✓	✗	4	✓	✗
21	Index from memory, address inter-procedural, result	<i>as above</i>	4	3	2	4	3	✓	✗	2	1	✗	✗	4	✓	✗	4	✓	✗
22	Index from memory, double indirect load	<i>as above</i>	3	2	1	3	2	✓	✗	1	0	✗	✗	3	✓	✗	3	✓	✗
23	Index from memory, memory is mutable	$C_{\text{MemoryMutable}}, C_{\text{TableIndirection}}$	3	2	1	3	2	✓	✗	1	0	✗	✗	3	✓	✗	3	✓	✗
24	C++ virtual calls from unrelated classes	$C_{\text{MultiPL}}, C_{\text{LowLevelTypes}}, C_{\text{MemoryMutable}}, C_{\text{TableIndirection}}$	23	23	20	23	24	✓	✗	19	16	✗	✗	23	✓	✓	23	✓	✓
<i>Total</i>			107	80	56	87	68	21	8	60	35	7	6	77	18	7	92	22	9

functions by traversing the graph, starting from the entry points. Other analyses only produce the set of reachable functions and no call graph; in such cases, we compare the sets of reachable functions. For testing precision, all but the last benchmark contain at least one non-reachable function that an analysis should identify as such.

The first three benchmarks contain simple direct calls and should pose no challenge. Benchmarks 4 to 8 test whether an analysis correctly determines implicit entry points and which functions are reachable from the host. All benchmarks from 7 on contain at least one indirect call and hence involve $C_{\text{TableIndirection}}$ (omitted from the table for brevity). Benchmarks 6 to 11 contain JavaScript host code that, if analyzed, can improve the precision of the call graph. In contrast, the call graph of benchmarks 12 to 23 can be determined

precisely from only the binary. They deal with increasingly more complex expressions for the table index passed to an indirect call ($C_{\text{TableIndexValue}}$). Benchmarks 12 to 14 all have a constant index value, whereas later ones require tracking of data-flow through locals and functions, and from benchmark 19 on also through memory. The last benchmark is compiled from a small C++ program with Emscripten, the most popular compiler targeting WebAssembly. The program contains two unrelated class hierarchies and two virtual calls. Knowledge about the source semantics could help an analysis to improve the precision of the targets of the virtual calls.

4.2 Existing Call Graph Analyses

We evaluate the call graph analyses implemented in four tools: Wassail [45], MetaDCE (from the Binaryen tool suite) [7], Twiggy [13], and WAVM in combination with an LLVM call graph analysis [2, 10].

Wassail is a research static analysis toolkit for WebAssembly binaries. It can produce an explicit call graph for a given binary, e.g., as a .dot file. MetaDCE and Twiggy statically analyze a given binary for the purpose of optimization, in particular binary debloating. MetaDCE takes as input a binary and a list of reachability roots (i.e., entry points). It then computes a reachability graph and removes all unreachable functions. Twiggy is a code size profiler that also constructs a reachability graph. It reports which parts of the binary are reachable from the outside, and how much space could be saved by removing them. MetaDCE and Twiggy do not explicitly produce a call graph, unlike Wassail, so we compare them in terms of the set of reachable functions F_r only.

Another approach for extracting a call graph from a WebAssembly binary is to lift it to a common intermediate representation, such as LLVM IR, for which an existing call graph analysis can be reused. The aWasm compiler [6] lifts WebAssembly to LLVM IR and C code. Unfortunately, it does not support all WebAssembly features yet and crashes on a majority of real-world binaries, whenever a table is either exported or the number of entries in the table are greater than 1,024. Because of this, we do not consider it in our study. WAVM is another compiler that lifts WebAssembly to LLVM IR, to subsequently generate machine code with near native performance. We invoke WAVM to produce LLVM IR from a given WebAssembly binary and then extract a call graph from the IR via LLVM’s `opt` tool with the `-dot-callgraph` option.

4.3 Experimental Setup and Results

For each microbenchmark, we compare the call graph (for Wassail and WAVM+LLVM) and the set of reachable functions (for all four analyses) against our manually determined ground truth. If any call graph edge or reachable function is missing from the output, an analysis is **unsound** (✗ in the **S** column). If the analysis reports edges or reachable functions that can never be executed, the analysis is **imprecise** (✗ in the **P** column).

We will consider *soundness* first, as an unsound analysis can be very problematic for downstream applications. For example, in debloating, an unsound call graph will cause functions to be removed that are actually executed. Surprisingly, none of the existing analyses is fully sound. Wassail is unsound on benchmarks 7 and 8, because it misses that functions in an exported or imported table are reachable from the host, even if the functions are not exported themselves. WAVM+LLVM is unsound for the same reason, as is MetaDCE for benchmark 7 and Twiggy for benchmark 8. MetaDCE fails to identify the implicit WASI entry point on benchmark 5, and in benchmark 6 it assumes that an imported function can call any exported function and does not make the exported functions transitively reachable. MetaDCE is also unsound in benchmark 9, because it does not consider that host code can mutate an exported table and thus change the referenced functions at runtime.

Insight. One common source of unsoundness is WebAssembly’s interaction with the host environment, e.g., which affects what functions are reachable and whether the table may be mutated.

Additionally, WAVM+LLVM is unsound on all benchmarks from 11 onwards. This is because the LLVM call graph analysis cannot handle any of the indirect calls that were generated by WAVM from WebAssembly’s `call_indirect` instruction.

Insight. Even though it is possible to lift WebAssembly to LLVM IR for compilation, this step loses crucial information for analysis.

Implication. Static analysis should likely be performed directly on the WebAssembly bytecode.

Besides unsoundness, Wassail, MetaDCE, and Twiggy also *crash* for some benchmarks, even though the binaries are standards-compliant. Wassail (MetaDCE) crash on benchmark 10 (10 and 11), where the table (memory) initialization offset is imported from the host. Twiggy crashes in the presence of a WebAssembly `start` section in benchmark 4. Such a crash may prevent developers from size-profiling their binaries, and thus prevent them from reducing binary bloat. The Twiggy authors have confirmed our crashes and are working on fixing them.

Insight. Our microbenchmarks uncover several sources of unsoundness and even causes existing call graph analyses to crash.

Implication. Future analyses should test against our benchmarks to avoid common mistakes.

Finally, our benchmarks also uncover different sources of *imprecision*. All existing analyses fail to take data-flow into account (through locals and inter-procedurally) and lack pointer analysis to precisely determine the targets of indirect calls in benchmarks 14 to 23, which leads to lackluster overall precision of all analyses. Wassail is the only one that constrains call targets of indirect calls by their type signature, which gives it a precise result for benchmark 13. MetaDCE and Twiggy choose a particularly conservative and thus imprecise approach to analyzing the table section in general. They mark every function as reachable that is present in an *elem* section. They neither use the index value nor type of indirect calls to further constrain the targets of indirect calls.

Insight. All existing analyses are imprecise, and in some cases ignore information that is available in the binary.

Implication. Future analyses should (i) use types to constrain indirect call targets, (ii) track data-flow and memory, and (iii) analyze host code to improve precision.

5 EVALUATION ON REAL-WORLD BINARIES

Here, we investigate whether existing call graph analyses are sound when applied to real-world WebAssembly binaries. We collect binaries and test inputs from real-world projects (Section 5.1) to obtain sets of dynamically executed functions, which we then compare against the reachable functions as reported by the static analyses. The results are summarized in Table 2 and detailed in Section 5.3.

5.1 Real-World Binaries

Unfortunately, it is impractical to manually produce ground truth call graphs for large binaries with thousands of functions. Therefore,

Table 2: Overview of the real-world programs.

Name	Source	LoC	Binary Size in KB	Number of Tests	Function Cov. of the Tests
sql.js	C	165,491	1,100	2	31.05–31.86%
opencv	C++	973,964	7,000	4	5.85–6.31%
rsa	Rust	20,849	369	1	35.13%
blake	Rust	20,345	35	2	21.25–26.25%
libmagic	C++	17,238	290	2	31.69%
graphviz	C++	857,121	929	2	31.02–39.72%
source-map	Rust	1809	48	2	20.00–40.00%
shiki	C/C++	86,316	467	3	40.67–49.28%
font-editor	C/C++	39,417	727	3	4.66–35.45%
opusscript	C/C++	53,898	276	3	49.70–50.30%

we collect the dynamically reachable functions during execution as a baseline for soundness (but not for precision, as we cannot know which inputs might make further functions reachable).

The WasmBench dataset we use in Section 3.2 is ill-suited for this task. It contains only binaries and metadata, but neither host code nor test cases to execute the binaries. We thus collect ten real-world binaries ourselves. As WebAssembly is most popular in JavaScript host environments, we collect them from NPM with these criteria: (i) The package must contain WebAssembly code without language extensions. We query the NPM registry with the keywords `wasm` and `WebAssembly` and manually inspect the resulting packages. (ii) We select the most popular packages, for which NPM provides an explicit ranking. (iii) We require packages with test cases or (for libraries) with client code exercising the library, which we find through manual inspection. (iv) We select binaries that were compiled from different source languages and toolchains.

The binaries are shown in Table 2. All of them are widely used, often originally native libraries that were compiled to WebAssembly. SQL.js is the well-known SQLite database; OpenCV and Graphviz the eponymous C++ libraries; two Rust projects implement the RSA and Blake cryptographic primitives; libmagic is known from the `file` UNIX utility to determine file types; `source-map` is a library from Mozilla for source mapping in Node.js and web applications; Shiki is a syntax highlighter that uses the `oniguruma` regex library; Fonteditor contains a library for the `woff2` font format; OpusScript is a port of the common Opus audio codec. We exercise each binary with several test cases, either taken from the repositories or implemented ourselves following the documentation. Each test performs a larger task, e.g. for SQL.js, loading a database, querying and modifying several tables, and saving it again. The rightmost column shows how many functions of the binary are exercised in the tests. As we can see from the OpenCV example, very large projects are compiled to WebAssembly, but only a small fraction of their functions may be used in downstream applications. This motivates binary debloating, which in turn requires call graph analysis.

5.2 Experimental Setup

We evaluate the same analyses as in Section 4.2. Since MetaDCE requires a list of entry points as input, we take all exported functions that are executed by at least one of the test cases, as determined by dynamic analysis. In this experiment, we focus on sets of reachable

functions, so for Wassail and WAVM+LLVM, we obtain those by traversing the call graph, starting from the dynamic entry points. MetaDCE and Twiggy directly return unreachable functions.

We obtain the set of dynamically executed functions with the Wasabi framework [31], which instruments each binary with a small dynamic analysis. Besides for the entry points, we use this data to determine all executed functions for each test case. The union of the executed functions for all test cases of a binary is F_{dyn} .

As in the previous section, F_{all} is the set of all functions in a binary. For each static analysis, we report the size of the set of functions that it deems reachable, $|F_r|$. Conversely, we also report the set of functions that can be removed from the binary according to the analysis as $F_{\text{del}} = F_{\text{all}} - F_r$. Ideally, this should be as large as possible. However, at the same time, no function should be removed that is actually used, as this would introduce unsoundness. We quantify unsoundness with the set $F_{\text{unsound}} = F_{\text{dyn}} - F_r$.

5.3 Results

In Table 3, we see to what extent the soundness and precision limitations of the analyses affect real-world binaries. At a high level, we see that three out of four analyses are not sound, as they (propose to) remove functions that are actually executed. While MetaDCE does not incorrectly remove functions, it crashes on the largest real-world program, so clearly no analysis is perfect.

Both Wassail and Twiggy assume that the call graph analysis of a WebAssembly binary is done separately from the host code that it interacts with. In contrast, MetaDCE takes as an input a set of initially reachable nodes in the call graph. While for this evaluation, we have not analyzed the corresponding host code, a call graph of host code can be passed to MetaDCE, which would be useful when performing dead-code elimination on a WebAssembly binary.

We see unsoundness in all the analyses on the `opencv` library. MetaDCE crashes when analyzing `opencv` which is because it makes the assumption that the element and data sections of the WebAssembly binary are initialized with a constant. All other analyses are unsound because they make the assumption that functions in an exported table are not reachable by the host. However, in the case of `opencv`, the binary exports its table and the tests call functions that are present in the table, even though they are not exported themselves. Wassail also shows unsoundness in the case of `graphviz` and `opusscript`. This is because the host code in `graphviz` calls functions from the exported table. WAVM+LLVM is unsound for all but two real-world programs. We see a high number of functions being reported as unreachable that are in fact reachable, because the `opt` tool does not evaluate indirect calls, as discussed in Section 4.3.

Besides soundness, we also see that Twiggy and MetaDCE are more conservative with removing functions, then Wassail. This could be because MetaDCE and Twiggy mark all functions in element sections as reachable, as discussed in Section 4.3.

Table 3: Evaluation of the soundness of existing call graph analyses on real-world programs. We report the number of incorrectly ($F_{\text{unsound}} = F_{\text{dyn}} - F_r$) and correctly removed functions ($F_{\text{del}} = F_{\text{all}} - F_r$). Unsoundness is highlighted.

Library	$ F_{\text{all}} $	$ F_{\text{dyn}} $	Wassail			WAVM+LLVM			MetaDCE			Twiggy		
			$ F_r $	$ F_{\text{unsound}} $	$ F_{\text{del}} $	$ F_r $	$ F_{\text{unsound}} $	$ F_{\text{del}} $	$ F_r $	$ F_{\text{unsound}} $	$ F_{\text{del}} $	$ F_r $	$ F_{\text{unsound}} $	$ F_{\text{del}} $
sql.js	1261	390	1257	0 (0%)	4 (0%)	633	124 (32%)	628 (50%)	1261	0 (0%)	0 (0%)	1261	0 (0%)	0 (0%)
opencv	10909	684	Timeout			822	477 (70%)	10087 (92%)	Crash			822	477 (70%)	10087 (92%)
rsa	785	273	777	0 (0%)	8 (1%)	564	3 (1%)	221 (28%)	785	0 (0%)	0 (0%)	785	0 (0%)	0 (0%)
blake	81	21	76	(0%)	5 (6%)	57	(0%)	27 (30%)	81	(0%)	(0%)	81	(0%)	(0%)
libmagic	736	218	716	0 (0%)	20 (3%)	88	152 (70%)	648 (88%)	736	0 (0%)	0 (0%)	736	0 (0%)	0 (0%)
graphviz	2018	790	2006	12 (2%)	17 (1%)	569	357 (45%)	1447 (72%)	2018	(0%)	(0%)	2018	(0%)	(0%)
source-map	46	19	38	0 (0%)	8 (17%)	38	0 (0%)	8 (17%)	46	0 (0%)	0 (0%)	46	0 (0%)	0 (0%)
shiki	213	105	213	(0%)	(0%)	154	11 (17%)	57 (28%)	213	(0%)	(0%)	213	(0%)	(0%)
fonteditor	1118	381	1118	0 (0%)	0 (0%)	345	306 (81%)	773 (69%)	Crash			345	306 (81%)	773 (69%)
opuscript	356	169	49	127 (75%)	307 (86%)	49	127 (75%)	307 (86%)	356	(0%)	(0%)	356	(0%)	(0%)

Insight. The unsoundness of current static call graph analyses manifests itself in real-world binaries, which may cause incorrect optimizations and other problems in downstream applications. Different tools face different shortcomings. E.g., because WAVM builds on LLVM, it loses vital information from the original bytecode and thus is unfit for sound call graph analysis. MetaDCE and Binaryen suffer less from unsoundness but also remove many fewer functions, i.e., they are more sound at the cost of precision.

6 THREATS TO VALIDITY

The results and conclusions of this study are subject to potential threats to validity, which we have tried to mitigate in the following ways: First, not all of the studied analyses explicitly construct call graphs. However, all target closely related tasks, such as obtaining sets of reachable functions. As a common denominator across all analyses, we compare them based on the set of reachable functions, which still relies on their ability to construct accurate call graphs. Second, to extract the call graphs and reachable functions computed by the existing analyses, we process the outputs produced by the studied tools. We carefully check our code to accurately extract the results of the analyses, but cannot fully exclude the possibility of accidentally misrepresenting their abilities. Third, our set of microbenchmarks is designed to cover particularly challenging language features. The effectiveness of an analysis on these microbenchmarks may not translate proportionally to real-world binaries, as not all challenges are equally prevalent in practice. For that reason, we did evaluate the analyses additionally on a set of real-world binaries. Fourth, to assess the soundness of static call graphs in real-world programs, we use dynamic call graph analysis driven by test inputs. Any bias in those inputs, e.g., favoring particular language features, may impact the results of our study on real-world binaries. Except for OpenCV, the union of our tests however exercises a large fraction of the binaries, at least one third to half of all functions. Finally, all results are limited to WebAssembly, and do not allow for drawing conclusions about other languages.

7 RELATED WORK

WebAssembly in general. Since the initial publication that introduced the language [26], WebAssembly has received significant interest by researchers. For example, there is work on formalizing and improving the WebAssembly type system [52], studying its performance in comparison to native binaries [29], and understanding its security implications [30]. WasmBench offers thousands of real-world WebAssembly binaries, which are used to study their usage in the wild [28].

Program analysis of WebAssembly. In addition to the call graph analyses studied in this paper, various program analyses for WebAssembly have been proposed. Static analyses include slicing [44], predicting higher-level types [32], an outline of a static analysis library [45], a vulnerability detector [20], and a code transformation technique that tries to hide malicious JavaScript code by translating parts of it to WebAssembly [41]. Concurrently with our work, Paccamiccio et al. propose a static call graph analysis based on symbolic execution and slicing [36]. As their implementation has not been available while conducting our study, we do not include it in our study. On the dynamic analysis side, there is work on taint tracking [23, 48] and fuzzing [27]. Wasabi [31] offers a general framework for implementing dynamic analyses. Given the relatively young age of the language, we expect to see many more analyses in the future, and accurate static call graphs could serve as a basis for them.

Call graph construction. Due to the fundamental nature of call graphs as a building block for many downstream program analyses, various algorithms for statically constructing call graphs exist. RTA [19] and XTA [49] target object-oriented languages. Other algorithms specifically target Java libraries and different usage scenarios of them [39], analyze Python code [43], adapt existing call graph algorithms to Scala [17], or address the related problem of call chain analysis, i.e., whether specific call stacks are feasible [42]. Motivated by the inevitable imprecision of sound call graph analysis, Utture et al. propose a learning-based pruning of static call graphs [51]. When analyzing JavaScript-based web applications, static call graph construction often is unsound-by-design [22]. Nielsen et al. target

server-side JavaScript with a modular analysis of individual packages [35]. To support multi-language web applications, a static analysis of server-side PHP code approximates the call graph of client-side JavaScript code generated by PHP [34]. Besides static analyses, dynamic call graph construction also poses some challenges, e.g., due to the interaction of applications with complex frameworks [54].

Studies of call graph analyses. Several studies empirically compare different call graph analyses with each other. Murphy et al. compare nine analyses for C [33]. Grove et al. study the precision and cost of algorithms for object-oriented languages [25] and describe different algorithms in a unified framework [24]. Studies by Reif et al. [40] and Sui et al. [47] compare analyses for Java. Another study targets different languages on the JVM [16]. Chakraborty et al. propose to identify the root causes of missing call graph edges via dynamic analysis [21]. To the best of our knowledge, this paper is the first comprehensive study of call graph analyses for WebAssembly, including an analysis of its unique challenges and a novel benchmark based on them.

Applications of call graphs. Static call graphs are a basic ingredient of various inter-procedural analyses. Examples include static debloating, e.g., of C/C++ [37, 38], binary shared libraries [15] and JavaScript [50], fault localization based on stack traces [53], and static taint analysis, e.g., of Android apps [18]. We envision our work to help improve call graph analyses for WebAssembly, which will ultimately benefit downstream inter-procedural analyses.

8 CONCLUSIONS AND RECOMMENDATIONS

As WebAssembly is becoming increasingly important, the language is a highly relevant target for static analysis. One of the most fundamental static analyses is call graph construction, which serves as a building block for many inter-procedural analyses and provides value on its own, e.g., for binary debloating. This paper presents the first systematic study of challenges for constructing call graphs in WebAssembly, how prevalent those challenges are in real-world binaries, and how those challenges are handled by current static analyses. Surprisingly, we find that all studied analyses are unsound and suffer from imprecision. These limitations are caused by WebAssembly-specific challenges, such as indirect calls controlled via a mutable function table, interactions between WebAssembly and its host environment, and unmanaged linear memory. We also show that the issues are not just limited to corner cases, but affect large, real-world binaries. To help improve future analyses, we also provide a set of executable microbenchmarks to test for specific challenges.

Our work has several implications and recommendations for future call graph analyses. First, we recommend operating on the level of WebAssembly bytecode, not an intermediate representation such as LLVM IR, in order to not lose information that is crucial for precision. Second, readily available information from the binary should be taken into account by call graph analyses, such as the type information in indirect calls, which could constrain possible call targets. Third, the precision of the call graph relies critically on the ability of the analyses to reason about local and inter-procedural data flow, and on pointer analysis, which currently no analysis

incorporates. Finally, future analyses could further improve precision by going beyond just the binary, e.g., taking into account the host code of an application, or the source language semantics from which it was compiled. One challenge for the latter idea will be that not every source-level function is represented as one WebAssembly function, e.g., due to optimizations and inlining.

9 ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), by the German Research Foundation within the ConCSys and DeMoCo projects, and by National Science Foundation grant CCF-1907727.

REFERENCES

- [1] [n. d.]. WASI – The WebAssembly System Interface. <https://wasi.dev/>
- [2] [n. d.]. WAVM. <https://wavm.github.io/>
- [3] Andreas Rossberg (Ed.). [n. d.]. *WebAssembly Core Specification*. World Wide Web Consortium (W3C). <https://www.w3.org/TR/wasm-core-1/>
- [4] 2020. *Wasmtime – A small and efficient runtime for WebAssembly & WASI*. <https://wasmtime.dev/>
- [5] 2020. *WebAssembly*. <https://webassembly.org/>
- [6] 2022. *aWasm - An Awesome Wasm Compiler and Runtime*. <https://github.com/gwsystems/awasm/>
- [7] 2022. Binaryen. See <https://github.com/WebAssembly/binaryen>.
- [8] 2022. Clang: a C language family frontend for LLVM. Available from <https://clang.llvm.org/>.
- [9] 2022. Introduction to WebAssembly using Go. Available from <https://golangbot.com/webassembly-using-go/>.
- [10] 2022. The LLVM Compiler Infrastructure. Available from <https://llvm.org/>.
- [11] 2022. Node.js with WebAssembly. Available from <https://nodejs.dev/en/learn/nodejs-with-webassembly>.
- [12] 2022. Rust: A language empowering everyone to build reliable and efficient software. Available from <https://www.rust-lang.org/what/wasm>.
- [13] 2022. Twiggy: A code size profiler for Wasm. See <https://rustwasm.github.io/twiggy/>.
- [14] 2022. WABT: The WebAssembly Binary Toolkit. See <https://github.com/WebAssembly/wabt>.
- [15] Ioannis Agadakov, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 70–83.
- [16] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2021. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2644–2666. <https://doi.org/10.1109/TSE.2019.2956925>
- [17] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2014. Constructing call graphs of Scala programs. In *European Conference on Object-Oriented Programming*. Springer, 54–79.
- [18] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
- [19] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 324–341.
- [20] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. [n. d.]. Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly. ([n. d.]), 102745. <https://doi.org/10.1016/j.cose.2022.102745>
- [21] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hasmanshahi. 2022. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:28. <https://doi.org/10.4230/LIPIcs.ECOOP.2022.3>
- [22] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 752–761.
- [23] William Fu, Raymond Lin, and Daniel Inge. [n. d.]. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. [abs/1802.01050](https://arxiv.org/abs/1802.01050) ([n. d.]). [arXiv:1802.01050](https://arxiv.org/abs/1802.01050) [cs.CR]

- [24] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.
- [25] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-Oriented Languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.
- [26] 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* (Barcelona, Spain) (PLDI 2017). ACM, New York, NY, USA, 185–200. <https://doi.org/10.1145/3062341.3062363>
- [27] Keno Haßler and Dominik Maier. 2021. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-Oriented Trends Symposium (Vienna, Austria) (ROOTS'21)*. Association for Computing Machinery, New York, NY, USA, 23–30. <https://doi.org/10.1145/3503921.3503924>
- [28] 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 (Ljubljana, Slovenia) (WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. <https://doi.org/10.1145/3442381.3450138>
- [29] 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)*. USENIX Association, Renton, WA, 107–120. <https://www.usenix.org/conference/atc19/presentation/jangda>
- [30] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–217. <https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann>
- [31] 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* (Providence, RI, USA) (ASPLOS '19). Association for Computing Machinery, New York, NY, USA, 1045–1045. <https://doi.org/10.1145/3297858.3304068>
- [32] 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* (New York, NY, USA) (PLDI '22). Association for Computing Machinery. <https://doi.org/10.1145/3519939.3523449>
- [33] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 158–191.
- [34] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 518–529.
- [35] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 29–41. <https://doi.org/10.1145/3460319.3464836>
- [36] Mattia Paccamiccio, Franco Raimondi, Kelly Androutsopoulos, and Leonardo Mostarda. 2022. Improving the Precision of Call Graph Construction for Webassembly with Symbolic Execution and Slicing. <http://dx.doi.org/10.2139/ssrn.4341186>.
- [37] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 164–180. <https://doi.org/10.1145/3385412.3386017>
- [38] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through {Piece-Wise} Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. 869–886.
- [39] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for Java libraries. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 474–486. <https://doi.org/10.1145/2950290.2950312>
- [40] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 251–261. <https://doi.org/10.1145/3293882.3330555>
- [41] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022)*. IEEE Computer Society, 1101–1101. <https://doi.org/10.1109/SP46214.2022.00064>
- [42] Atanas Rountev, Scott Kagan, and Michael Gibas. 2004. Static and dynamic analysis of call chains in Java. In *International Symposium on Software Testing and Analysis (ISSTA)*.
- [43] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1646–1657. <https://doi.org/10.1109/ICSE43902.2021.00146>
- [44] Quentin Stiévenart, Dave Binkley, and Coen De Roover. 2022. Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries. In *The 44th International Conference on Software Engineering (ICSE 2022)*. <https://doi.org/10.1145/3510003.3510070>
- [45] Quentin Stiévenart and Coen De Roover. [n. d.]. Wassail: a WebAssembly Static Analysis Library (ProWeb21). <https://2021.programming-conference.org/home/proweb-2021> Fifth International Workshop on Programming Technology for the Future Web.
- [46] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 13–24. <https://doi.org/10.1109/SCAM51674.2020.00007>
- [47] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1049–1060. <https://doi.org/10.1145/3377811.3380441>
- [48] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. [n. d.]. Taint Tracking for WebAssembly. [abs/1807.08349](https://arxiv.org/abs/1807.08349) ([n. d.]). [arXiv:1807.08349](https://arxiv.org/abs/1807.08349)
- [49] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 281–293.
- [50] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. 2021. Stubbifier: Debloating Dynamic Server-Side JavaScript Applications. [CoRR abs/2110.14162](https://arxiv.org/abs/2110.14162) (2021). [arXiv:2110.14162](https://arxiv.org/abs/2110.14162) <https://arxiv.org/abs/2110.14162>
- [51] Akshay Uttre, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *ICSE*.
- [52] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (Los Angeles, CA, USA) (CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. <https://doi.org/10.1145/3167082>
- [53] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: locating crashing faults based on crash stacks. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 204–214. <https://doi.org/10.1145/2610384.2610386>
- [54] Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu. 2017. RunDroid: recovering execution call graphs for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 949–953. <https://doi.org/10.1145/3106237.3122821>