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

Anonymous author(s)

***Abstract*—WebAssembly is a low-level bytecode that powers applications and libraries running in browsers, on the server side, and in standalone runtimes. To understand, analyze, and optimize WebAssembly binaries, static call graph analysis is a central building block, e.g., for many inter-procedural analyses and binary debloating. However, statically constructing call graphs faces some unique challenges in WebAssembly. 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 WebAssembly-specific challenges, measure their prevalence in real-world binaries, and encode them into a suite of 24 executable microbenchmarks. The challenges include idiosyncracies of WebAssembly, such as indirect calls via a mutable function table, interactions with the host environment, and an unmanaged linear memory. We show that they commonly occur across a set of more than 8,000 real-world binaries. For example, 64% of indirect calls retrieve their targets from untyped, always mutable memory. Based on our microbenchmarks and a set of executable real-world binaries, we then study the soundness and precision of four existing industrial and academic 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 a reference for improved static call graph construction for WebAssembly. In particular, the benchmarks presented in this paper will allow future work to check whether an analysis supports challenging language features, and to quantify its soundness and precision.**

I. Introduction

WebAssembly [1], [2] 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 [3]. Today, WebAssembly is—as envisioned—widely used on the client-side, but also increasingly popular on the server-side, both in Node.js [4] and in standalone runtimes, e.g., Wasmtime [5]. 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. Several widely used industry tools, WABT [6], Binaryen [7], and Twiggy [8], use static analysis to detect optimization opportunities. Stievenart et al. presented a static information flow analysis for detecting security vulnerabilities [9], a static slicing technique for WebAssembly with applications in reverse engineering, code comprehension, and security [10], and the design of a general-purpose static analysis framework for WebAssembly [11].

A particularly important kind of analysis is static call graph construction. Call graphs are at the core of many inter-procedural analyses. Moreover, a call graph is also useful by itself, e.g., removing unused code 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.

However, statically computing a sound and precise call graph is challenging. While this is true for all programs due to Rice's theorem, there are also a number of challenges that are specific to certain languages and program representations. For example, in WebAssembly, indirect function calls are implemented by indexing into a function table, 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 [12] or Java [13], [14], 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 effect 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 WebAssembly-specific challenges, measure their prevalence, and evaluate the call graphs produced by four existing analyses: Wassail [11], the LLVM-based WAVM [15], MetaDCE from the Binaryen tool suite [7], and Twiggy [8]. Our study addresses the following research questions.

- **Which WebAssembly language constructs pose challenges for static call graph construction and how commonly are they used in real-world programs?**
  To answer this question, we introduce 12 challenges grouped into six categories, and measure their prevalence on a large set of more than 8,000 real-world WebAssembly binaries [3]. (Section III)
- **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 existing analyses. (Section IV)

- **To what extent are existing analyses sound for large, real-world binaries?** To answer this question, we dynamically analyze a set of six real-world binaries and then compare the executed functions to the call graphs and reachable functions determined by the static analyses. (Section V)

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. In three of four analyses, functions are considered unreachable, even though these functions are actually executed in test cases. Several of our microbenchmarks and one of the real-world programs also lead to crashes in the existing analyses tools.

In summary, this paper contributes the following:

- The first systematic categorization of the WebAssembly-specific challenges of static call graph construction.
- Measurements how prevalent those challenges are in real-world binaries.
- A set of microbenchmarks and ground-truth call graphs reflecting those challenges, which can be used to assess the soundness and precision of existing analyses.
- An evaluation of existing call graph analyses, both on microbenchmarks and on 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://anonymous.4open.science/r/wasm-call-graphs-B43D/.

II. Background on WebAssembly

We give a short introduction to the WebAssembly language and ecosystem. For more details, please refer to the website [1], the initial academic publication [2], and the official language specification [16]. 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 [3]. 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 [4] and on standalone runtimes, e.g., Wasmtime [5]. In this case, WASI [17] 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++ [18], [19], Rust [20], and Go [21]. Unless otherwise specified, this paper refers to WebAssembly 1.0, which is the current version of the language, and supported by all major browsers and stand-alone runtimes.

```c
int square(int x) { return x * x; }
size_t strlen(const char *s) { ... }
```

compiled with Emscripten or Clang to:

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

Fig. 1: Simple example of C code compiled to WebAssembly.

Figure 1 shows a simplified WebAssembly program compiled from C. A WebAssembly *module* contains a set of *functions*, where each function takes a sequence of values as parameters and produces a sequence 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 the *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 typed language, but supports only four low-level types: 32-bit integers (`i32`), 64-bit integers (`i64`), 32-bit floating point values (`f32`), and 64-bit floating point values (`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 to the host environment once the module has been instantiated. The *instantiation* of a module checks that the provided imports match the declared types and performs

Program representation:

$\mathbf{C}_{\textbf{FunctionNames}}$	No function names, only indices.
$\mathbf{C}_{\textbf{ProgramStructure}}$	Low-level program structure, static linkage.

Indirect calls and table section:

$\mathbf{C}_{\textbf{TableIndirection}}$	Double indirection in indirect calls requires knowledge about table contents.
$\mathbf{C}_{\textbf{TableIndexValue}}$	Table index values (function pointers) are hard to identify and determine statically.
$\mathbf{C}_{\textbf{TableMutation}}$	Table may be mutated by host code.
$\mathbf{C}_{\textbf{TableInitOffset}}$	Initialization may depend on host code.

Types:

$\mathbf{C}_{\textbf{LowLevelTypes}}$	Types are low-level and imprecise.

Host environment:

$\mathbf{C}_{\textbf{HostCallbacks}}$	Outgoing edges of imported functions.
$\mathbf{C}_{\textbf{EntryPoints}}$	No standard entry point(s).

Memory:

$\mathbf{C}_{\textbf{MemoryMgmt}}$	No built-in memory management.
$\mathbf{C}_{\textbf{MemoryMutable}}$	Linear memory is writable everywhere.

Source languages:

$\mathbf{C}_{\textbf{MultiPL}}$	Binaries are compiled from different source languages, there is no standard library.

Fig. 2: Overview of the WebAssembly-specific challenges.

initializations. After instantiation, the functions exported by 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.

III. Challenges and Their Prevalence

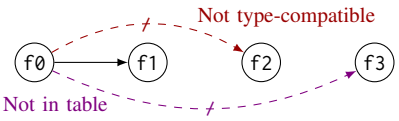
This section presents challenges for static call graph construction in WebAssembly (Section III-A) and their prevalence in real-world binaries in Section III-B. In total, we identify 12 WebAssembly-specific challenges, as summarized in Figure 2.

A. Challenges

1) Program Representation: Unlike in source code or JVM bytecode, functions in WebAssembly are not identified by names but with integer indices ($\mathbf{C}_{\textbf{FunctionNames}}$). For example, a direct call to the function with index 23 is just `call 23`. The index does not convey any semantics or intuition about the function, making it difficult to identify special-purpose functions, e.g., `memcpy`. A related challenge is WebAssembly's low-level program structure ($\mathbf{C}_{\textbf{ProgramStructure}}$). All functions are laid out in a single flat index space, without any structure to express source-level concepts such as scopes or classes. For example, whether a function is a method, and of which class, is not readily available. WebAssembly code is also statically linked, such that functions from different libraries cannot be easily distinguished.

```
1  (module
2    (func (;0;)
3      instruction sequence producing an i32 index...
4      ;; Calls the function at table index given above;
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)
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).

Fig. 3: Example of indirect calls in WebAssembly.

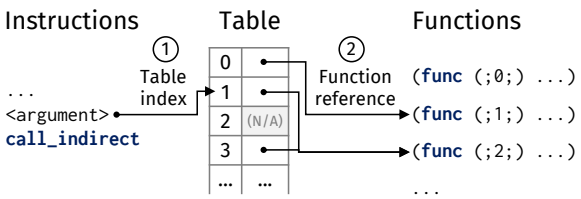


Fig. 4: Illustrating the double indirection through the table.

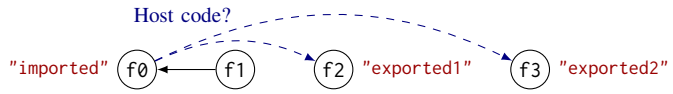
2) Indirect Calls and Table Section: Besides regular direct calls, WebAssembly has *indirect calls*, where the call target is determined at runtime. Those implement function pointers, virtual calls, and other forms of dynamic dispatch. WebAssembly has only a single instruction for this purpose, `call_indirect`. In the example of Figure 3, 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* ($\mathbf{C}_{\textbf{TableIndirection}}$), as illustrated in Figure 4. To determine an indirect call's potential targets, an analysis thus needs to reason about both index values and table contents.

Determining the index value of an indirect call is particularly challenging ($\mathbf{C}_{\textbf{TableIndexValue}}$) since index values are of type `i32`, i.e., indistinguishable from other pointers or numbers, and they can be manipulated like any integer. This differs from higher-level representations, where function pointers can be identified by type, and from native code, where code pointers can be identified because they point into code space.

We identify two challenges specific to determining a table's state. A table is initialized via the *elem* section (line 16 of Figure 3a), which copies function references into the table

```
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`.

Fig. 5: Example of imported and exported functions.

	Distribution	#Bin.	Finding	Challenge(s)
1	74%	6,239	Has no function `name` section	$\mathbf{C}_{\text{FunctionNames}}$
2	95%	8,014	Has no `.debug` section	$\mathbf{C}_{\text{FunctionNames}}$
3	83%	6,961	At least one indirect call	$\mathbf{C}_{\text{TableIndirection}}$
4	22%	1,846	Table is imported or exported	$\mathbf{C}_{\text{TableMutation}}$, $\mathbf{C}_{\text{HostCallbacks}}$
5	12%	1,084	Element offset from variable	$\mathbf{C}_{\text{TableInitOffset}}$
6	92%	7,680	At least one import function	$\mathbf{C}_{\text{HostCallbacks}}$
7	97%	8,144	No explicit `start` function	$\mathbf{C}_{\text{EntryPoints}}$
8	87%	7,339	No explicit WASI start function	$\mathbf{C}_{\text{EntryPoints}}$
9	95%	7,993	At least one store instruction	$\mathbf{C}_{\text{MemoryMutable}}$

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

starting from a specific offset. As the offset may be read from an imported global variable via `global.get`, determining the layout of the table is difficult ($\mathbf{C}_{\text{TableInitOffset}}$). To make matters worse, tables may be mutated at any point in time ($\mathbf{C}_{\text{TableMutation}}$). While this is not possible within WebAssembly itself, the host environment can do so, e.g., using the JavaScript function `WebAssembly.Table.set()`.

3) Types: WebAssembly's four low-level types are of limited help for constructing call graphs ($\mathbf{C}_{\text{LowLevelTypes}}$). On the upside, the `call_indirect` instruction statically encodes the type of the target function (line 6 in Figure 3a), which constrains the set of call targets (Figure 3b). 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 *`, `int`) get compiled to simply `i32`. The absence of richer types implies that type-based call graph algorithms, e.g., RTA/XTA [22], [23] are not applicable to WebAssembly.

4) Host Environment: WebAssembly binaries always execute within a host environment (e.g., a JavaScript application) so each WebAssembly binary is only a partial program and determining its *entry points* is a challenge ($\mathbf{C}_{\text{EntryPoints}}$). 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 ($\mathbf{C}_{\text{HostCallbacks}}$), as illustrated in Figure 5. By analyzing the binary alone, an analysis has no way of precisely determining whether such calls are possible.

5) Memory: Two challenges are associated with WebAssembly's *linear memory*. First, WebAssembly does not provide any garbage collection or other high-level memory management ($\mathbf{C}_{\text{MemoryMgmt}}$) so memory management must be done by functions compiled into the binary itself. I.e., `malloc` and `free` are just regular functions in the binary. Hence, call graph algorithms that reason about allocation sites or types of objects [22] are not easily applicable to WebAssembly. Second, WebAssembly differs from native code in that *all linear memory is writable everywhere* ($\mathbf{C}_{\text{MemoryMutable}}$); even data that typically does not change during execution, e.g., vtables and static data, is potentially mutable.

6) Source Languages: WebAssembly binaries are compiled from a variety of different source languages, including C, C++, Rust, Go, and AssemblyScript [3]. 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 ($\mathbf{C}_{\text{MultiPL}}$). On a related note, there is no single standard library, for which one could special-case call graph construction.

Insight 1. WebAssembly's idiosyncratic features (e.g., indirect calls, tables, low-level types, and mutable linear memory) pose many challenges for call graph analyses.

B. Prevalence of Challenges in the Wild

Next, we measure how prevalent those challenges are in real-world WebAssembly binaries.

1) Dataset and Setup: We base our analysis on the WasmBench dataset [3], 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, ignoring challenges that are general properties of the language, e.g., $\mathbf{C}_{\text{ProgramStructure}}$ and $\mathbf{C}_{\text{MemoryMgmt}}$.

2) Results: Figures 6, 7, and 8 depict the results. Figure 6 presents measurements on the proportion of binaries affected by certain properties, and which challenges they relate to. From the first two rows, we see that $\mathbf{C}_{\text{FunctionNames}}$ 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 $\mathbf{C}_{\text{TableIndirection}}$. We also see that 22% of all binaries have either an imported or exported table. In such cases, since the host can mutate the table ($\mathbf{C}_{\text{TableMutation}}$), the state of the table cannot be determined from the binary alone. Moreover, the fact that all functions

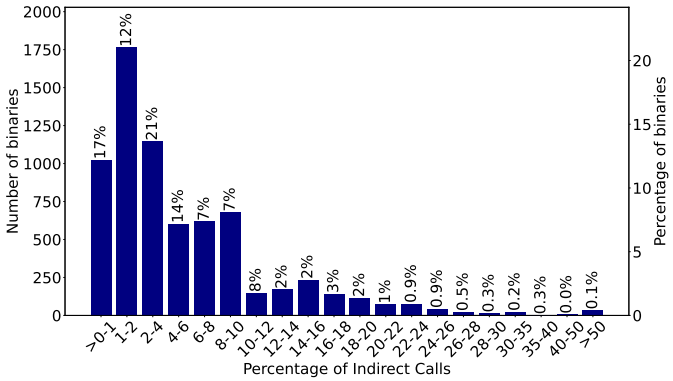


Fig. 7: Distribution of indirect calls in WasmBench binaries.

Distribution	Count	Description
0%	9.6k	Constant table index (`i32.const` n)
Data-flow related:		Index instructions involve...
49%	2.1M	local variable (`local.get/tee` l)
45%	1.9M	function parameter (`local.get/tee` p)
0%	16k	function result (`call` f/`call_indirect`)
Memory related:		Index instructions involve...
64%	2.7M	at least one `load`, whose address...
2%	96k	is a constant (`i32.const` n)
9%	364k	involves local variable (`local.get/tee` l)
13%	533k	involves parameter (`local.get/tee` p)
14%	586k	involves another load (`i32.load`)
58%	2.4M	involves some (pointer) arithmetic
Other:		
86%	3.5M	Index involves some arithmetic (see text)

Fig. 8: Instructions of the table index value ($\mathbf{C}_{\text{TableIndexValue}}$).

in an imported or exported table are reachable from the host may necessitate call graph edges that originate from imported functions ($\mathbf{C}_{\text{HostCallbacks}}$). Further, 12% of binaries initialize their table with an element section whose offset comes from an imported variable ($\mathbf{C}_{\text{TableInitOffset}}$), necessitating analysis of the host code to determine the (initial) state of the table.

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: average of 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 in Figure 6 but also related to $\mathbf{C}_{\text{HostCallbacks}}$ 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, potentially requiring pointer analysis for precision and suffering from WebAssembly's always writable memory.

Finally, we refer to Hilbig et al. [3] for their analysis of the source languages of WebAssembly binaries. 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, and 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.

Insight 2. The identified challenges impact many (sometimes even the majority) of the 8,392 binaries under consideration.

We dive more deeply into indirect calls in Figures 7 and 8, as they pose 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_indirects` in the dataset. Those instructions determine the index value ($\mathbf{C}_{\text{TableIndexValue}}$) and hence (together with the table) which function(s) may be called. The first row makes clear that most index values are not constants and thus not readily available. 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 the two. The first row shows that tracking data-flow through local variables is crucial, as 49% of index values involve them. 45% of index values require even inter-procedural data-flow analysis, since they involve the current function parameters or, to a much lesser degree, the return value of another function.

The next six rows analyze the involvement of memory in the instructions computing the index value. 64% of the preceding instruction sequences contain a load and hence would profit from 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. 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 3. Data-flow analysis (intra- and inter-procedural) and pointer analysis are crucial for increasing the precision of call graph analysis. The most important arithmetic operations are addition and bitmasking.

IV. Evaluating Soundness and Precision on Microbenchmarks

In this section, we evaluate the soundness and precision of existing static call graph analyses for WebAssembly. For that, we introduce a set of microbenchmarks (Section IV-A) that expose the challenges identified in Section III, and then evaluate four existing call graph analyses (Section IV-B) against these microbenchmarks. The results are summarized in Table I and detailed in Section IV-C.

A. Microbenchmarks

Table I 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 functions by traversing the graph, starting from the entry points. Other analyses only produce the set of reachable functions and no call graph. For those analyses, we do not compare their call graph. More details on the analyses follow in Section IV-B. 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 $\mathbf{C}_{\textbf{TableIndirection}}$ (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 ($\mathbf{C}_{\textbf{TableIndexValue}}$). 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.

B. Existing Call Graph Analyses

We evaluate the call graph analyses implemented in four tools: Wassail [11], MetaDCE (from the Binaryen tool suite) [7], Twiggy [8], and WAVM in combination with an LLVM call graph analysis [15], [18].

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 cannot be configured to specific entry points. It is a code size profiler that also constructs a reachability graph. It reports which parts of the binary are not 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 callgraph from a WebAssembly binary is to lift it to a common intermediate representation, such as LLVM IR, for which an existing callgraph analysis can be reused. The aWsm compiler [24] 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 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.

C. 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

TABLE I: 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. $|F_{\text{r}}|$ is the number of functions reachable from the given entry point(s). $|E|$ is the number of edges in the call graph. S indicates if the analysis is sound and P indicates if the analysis is precise, compared to the ground truth.

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

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. 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. This motivates more work on direct analysis of WebAssembly bytecode, without translating to another IR first. 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. It 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.

Besides unsoundness, Wassail, MetaDCE, and Twiggy also *crash* for some benchmarks, even though the binaries are standards-compliant. Wassail crashes on benchmark 10, where the table initialization offset is imported from the host.

MetaDCE crashes on benchmarks 10 and 11, where the table and memory initialization offset is imported from the host. Twiggy crashes in the presence of a WebAssembly `start` section in benchmark 4. The Twiggy authors have confirmed our crashes and are working on fixing them.

Finally, our benchmarks also uncover several sources of *imprecision* in the existing call graph analyses. In benchmark 10, Wassail does not analyze the imported global variable that is used to initialize the memory, which leads to imprecision. All tools fail to take data-flow into account (through locals and inter-procedurally) and lack pointer analysis to determine the targets of indirect calls in benchmarks 14 to 23. Wassail does however constrain call targets of indirect calls by their type signature (benchmark 13), which is a step in the right direction. 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 4. Our microbenchmarks uncover several sources of unsoundness and imprecision in existing call graph analyses, and even found several crashes. In particular, none of the considered analyses is sound and all of them are imprecise.

V. Evaluating Soundness on Real-World Binaries

In this section, we evaluate the soundness of existing call graph analyses on real-world WebAssembly binaries. For this task, we collect real-world binaries and test inputs for them (Section V-A) 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 II and detailed in Section V-C.

A. Real-World Binaries

Using the microbenchmarks of Section IV, we established the existence of unsoundness in several existing call graph analyses. However, this does not necessarily mean that such unsoundness manifests itself when applying these analyses to real-world binaries. Unfortunately, it is impractical to manually produce ground truth call graphs for large binaries with thousands of functions. Therefore, we execute them to automatically obtain sets of dynamically reachable functions, which we then use as a baseline for soundness (but not precision, as we cannot know which inputs might make further functions reachable).

Unfortunately, the WasmBench dataset we use in Section III-B is ill-suited for this task. It only contains binaries and metadata, but neither host code (e.g., JavaScript) nor test cases to execute the binaries. We thus collect six real-world binaries ourselves. As WebAssembly is most popular in JavaScript host environments, we collect them from NPM[1] using the following criteria: (i) The package must contain WebAssembly code and not use any language extensions. For that 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.

[1]https://www.npmjs.com/

TABLE II: Overview of the real-world programs.

Library	Source	LoC	KB	Test	Coverage
sql.js	C	165,491	1,100	#1	31.05%
				#2	31.86%
opencv	C++	973,964	7,000	#1	6.31%
				#2	5.85%
				#3	5.85%
				#4	5.85%
rsa	Rust	20,849	369	#1	35.13%
blake	Rust	20,345	35	#1	21.25%
				#2	26.25%
magic	C++	17,238	290	#1	31.69%
				#2	31.69%
graphviz	C++	857,121	929	#1	31.02%
				#2	39.72%

The resulting binaries are shown in Table II, with their source language, the lines of code (as determined by `cloc`), and size of the binary in KB. SQL.js is a WebAssembly port of the widely used SQLite database library. Similarly, OpenCV and graphviz are the eponymous C++ libraries compiled to WebAssembly. Two Rust projects implement the RSA and Blake cryptographic primitives, and the magic package is a port of libmagic, which is used, e.g., in the `file` UNIX utility to determine file types. We exercise each binary with several test cases. Tests are either taken from the developers or implemented ourselves following the documentation. Each test performs a larger task, e.g. for SQL.js, loading a database file, inserting, modifying, and querying several tables, and saving it again. The rightmost column shows the proportion of functions executed for each test. As we can see from the OpenCV example, very large projects are compiled to WebAssembly, but only a small fraction of their functions may actually be used in downstream applications. This motivates binary debloating, which in turn requires call graph analysis.

B. Experimental Setup

In terms of call graph analyses, we evaluate the same analyses as in Section IV-B. 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.

To determine the set of dynamically executed functions, we use the Wasabi dynamic analysis framework [25]. With it,

TABLE III: Evaluation of the existing call graph analyses on real-world program, in terms of soundness. The number of incorrectly removed functions ($F_{\text{unsound}} = F_{\text{dyn}} - F_{\text{r}}$) and correctly removed functions ($F_{\text{del}} = F_{\text{all}} - F_{\text{r}}$) are also reported. Unsoundness is highlighted in red and bold font.

Library	$\lvert F_{\text{all}}\rvert$	$\lvert F_{\text{dyn}}\rvert$	**Wassail**					**WAVM+LLVM**					**MetaDCE**					**Twiggy**				
			$\lvert F_{\text{r}}\rvert$	$\lvert F_{\text{unsound}}\rvert$		$\lvert F_{\text{del}}\rvert$		$\lvert F_{\text{r}}\rvert$	$\lvert F_{\text{unsound}}\rvert$		$\lvert F_{\text{del}}\rvert$		$\lvert F_{\text{r}}\rvert$	$\lvert F_{\text{unsound}}\rvert$		$\lvert F_{\text{del}}\rvert$		$\lvert F_{\text{r}}\rvert$	$\lvert F_{\text{unsound}}\rvert$		$\lvert F_{\text{del}}\rvert$	
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	822	**477**	**(70%)**	10087	(92%)	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	(0%)	5	(6%)	57	0	(0%)	24	(30%)	81	0	(0%)	0	(0%)	81	0	(0%)	0	(0%)
magic	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%)**	12	(1%)	569	**355**	**(45%)**	1449	(72%)	2018	0	(0%)	0	(0%)	2018	0	(0%)	0	(0%)

we instrument each binary and record all functions that are invoked during the execution of each test case. 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, $|F_{\text{r}}|$. Conversely, we also report the set of functions that can be removed from the binary according to the analysis as $F_{\text{del}} = F_{\text{all}} - F_{\text{r}}$. 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 $F_{\text{unsound}} = F_{\text{dyn}} - F_{\text{r}}$, and highlight all non-empty sets in Table II in red and bold.

C. Results

In Table III, 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 also 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 separate from the host code that it interacts with. In contrast, MetaDCE takes in as input a set of initially reachable nodes in the call graph. While for this evaluation, we have not analyzed the corresponding host code, MetaDCE is extensible to a host specific call graph, which would be useful when doing a host-code specific dead-code elimination analysis of 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. This is also because the host code in graphviz calls functions from the exported table. WAVM+LLVM is unsound for all but one real-world program. 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 IV-C.

Additionally, it is interesting to note that while both Twiggy and MetaDCE aim to reduce the code size of binaries, they are overly conservative. Comparatively, Wassail has a higher percentage of removed functions. This could be because MetaDCE and Twiggy mark all functions in element sections as reachable, as discussed in Section IV-C.

Insight 5. 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.

VI. Discussion

A. Broader Impact

Our study identifies important limitations in state of the art static call graph construction for WebAssembly. While having a perfectly sound and precise program analysis is generally impossible, call graph analyses should either strive for soundness, or at least, clearly document under what condition to expect unsoundness. On the positive side, three of the studied analyses (Wassail, MetaDCE, and Twiggy) address a relatively large fraction of the challenges encoded in our microbenchmarks. Given that WebAssembly, and hence also static analyses for it, are still at a relatively young age, we hope our work to contribute to further improvements of analyses we study and others to be written in the future. In particular, we envision our work to help improve call graph analyses (i) by making analysis writers aware of key challenges and (ii) by providing an executable microbenchmark to check to what extent an analyses addresses these challenges.

B. Threats to Validity

The results and conclusions of this study are subject to several threats to validity. First, not all of the studied analyses explicitly construct call graphs, but some target closely related

tasks, such as removing unreachable functions. As a common denominator across all analyses, we compare them based on the set of reachable functions, which only indirectly measures 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. Fourth, our results on the prevalence of challenges in real-world binaries, in particular the number of indirect calls and the distribution of instructions to compute table indices, may be skewed by the large binaries. Fifth, 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. Finally, all results are limited to WebAssembly, and do not allow for drawing conclusions about other languages. Likewise, our results on real-world binaries may or may not generalize to a broader set of binaries.

VII. Related Work

WebAssembly in general: Since the initial publication that introduced the language [2], WebAssembly has received significant interest by researchers. For example, there is work on formalizing and improving the WebAssembly type system [26], studying its performance in comparison to native binaries [27], and understanding its security implications [28]. WasmBench offers thousands of real-world WebAssembly binaries, which are used to study their usage in the wild [3].

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 [10], predicting higher-level types [29], and an outline of a static analysis library [11]. On the dynamic analysis side, there is work on taint tracking [30], [31] and fuzzing [32]. Wasabi [25] 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 [22] and XTA [23] target object-oriented languages. Other algorithms specifically target Java libraries and different usage scenarios of them [33], analyze Python code [34], adapt existing call graph algorithms to Scala [35], or address the related problem of call chain analysis, i.e., whether specific call stacks are feasible [36]. Motivated by the inevitable imprecision of sound call graph analysis, Utture et al. propose a learning-based pruning of static call graphs [37]. When analyzing JavaScript-based web applications, static call graph construction often is unsound-by-design [38]. Nielsen et al. target server-side JavaScript with a modular analysis of individual packages [39]. 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 [40]. Besides static analyses, dynamic call graph construction also poses some challenges, e.g., due to the interaction of applications with complex frameworks [41].

Studies of call graph analyses: Several studies empirically compare different call graph analyses with each other. Murphy et al. compare nine analyses for C [12]. Grove et al. study the precision and cost of algorithms for object-oriented languages [42] and describe different algorithms in a unified framework [43]. Studies by Reif et al. [13] and Sui et al. [14] compare analyses for Java. Finally, another study targets different languages on the JVM [44]. 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++ [45], [46], binary shared libraries [47] and JavaScript [48], fault localization based on stack traces [49], and static taint analysis, e.g., of Android apps [50]. We envision our work to help improve call graph analyses for WebAssembly, which will ultimately benefit downstream inter-procedural analyses.

VIII. Conclusion

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 and how these 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. Applying current analyses to real-world binaries shows them to be affected by the unsoundness problems, which may lead to unexpected behavior of downstream analyses. To remedy the current limitations of static call graph analyses, we provide a microbenchmark of WebAssembly binaries that test for specific challenges. We envision our work to guide future research on static call graph construction and static analysis for WebAssembly in general.

Data Availability

We make our benchmarks, datasets, source code, and results publicly available at https://anonymous.4open.science/r/wasm-call-graphs-B43D/.

REFERENCES

[1] (2020) Webassembly. [Online]. Available: https://webassembly.org/

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, pp. 185–200. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062363

[3] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, pp. 2696–2708.

[4] "Node.js with WebAssembly," 2022, available from https://nodejs.dev/en/learn/nodejs-with-webassembly.

[5] Wasmtime – a small and efficient runtime for webassembly & wasi. [Online]. Available: https://wasmtime.dev/

[6] "WABT: The WebAssembly Binary Toolkit," 2022, see https://github.com/WebAssembly/wabt.

[7] "Binaryen," 2022, see https://github.com/WebAssembly/binaryen.

[8] "Twiggy: A code size profiler for Wasm," 2022, see https://rustwasm.github.io/twiggy/.

[9] Q. Stiévenart and C. D. Roover, "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, 2020, pp. 13–24. [Online]. Available: https://doi.org/10.1109/SCAM51674.2020.00007

[10] Q. Stiévenart, D. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *The 44th International Conference on Software Engineering*, ser. ICSE 2022.

[11] Q. Stiévenart and C. De Roover, "Wassail: a webassembly static analysis library," ser. ProWeb21, fifth International Workshop on Programming Technology for the Future Web. [Online]. Available: https://2021.programming-conference.org/home/proweb-2021

[12] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[13] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "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*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 251–261. [Online]. Available: https://doi.org/10.1145/3293882.3330555

[14] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1049–1060. [Online]. Available: https://doi.org/10.1145/3377811.3380441

[15] WAVM. [Online]. Available: https://wavm.github.io/

[16] Webassembly core specification. World Wide Web Consortium (W3C). [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[17] Wasi – the webassembly system interface. [Online]. Available: https://wasi.dev/

[18] "The LLVM compiler infrastructure," 2022, available from https://llvm.org/.

[19] "Clang: a C language family frontend for LLVM," 2022, available from https://clang.llvm.org/.

[20] "Rust: A language empowering everyone to build reliable and efficient software," 2022, available from https://www.rust-lang.org/what/wasm.

[21] "Introduction to WebAssembly using Go," 2022, available from https://golangbot.com/webassembly-using-go/.

[22] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 324–341.

[23] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000, pp. 281–293.

[24] (2022) aWsm - An Awesome Wasm Compiler and Runtime. [Online]. Available: https://github.com/gwsystems/awsm/

[25] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, pp. 1045–1045.

[26] C. Watt, "Mechanising and verifying the webassembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, pp. 53–65.

[27] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *2019 USENIX Annual Technical Conference*, ser. USENIX ATC '19. Renton, WA: USENIX Association, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[28] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX Security 20. USENIX Association, Aug., pp. 217–217. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[29] D. Lehmann and M. Pradel, "Finding the dwarf: Recovering precise types from webassembly binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI '22. Association for Computing Machinery.

[30] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," vol. abs/1807.08349.

[31] W. Fu, R. Lin, and D. Inge, "TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly," vol. abs/1802.01050.

[32] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-Oriented Trends Symposium*, ser. ROOTS'21. New York, NY, USA: Association for Computing Machinery, 2021, p. 23–30. [Online]. Available: https://doi.org/10.1145/3503921.3503924

[33] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "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*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 474–486. [Online]. Available: https://doi.org/10.1145/2950290.2950312

[34] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "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, 2021, pp. 1646–1657. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00146

[35] K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip, "Constructing call graphs of scala programs," in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 54–79.

[36] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in Java," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[37] A. Utture, S. Liu, C. G. Kalhauge, and J. Palsberg, "Striking a balance: Pruning false-positives from static call graphs," in *ICSE*, 2022.

[38] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "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*, 2013, pp. 752–761.

[39] B. B. Nielsen, M. T. Torp, and A. Møller, "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*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 29–41. [Online]. Available: https://doi.org/10.1145/3460319.3464836

[40] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "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, 2014, pp. 518–529.

[41] Y. Yuan, L. Xu, X. Xiao, A. Podgurski, and H. Zhu, "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*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 949–953. [Online]. Available: https://doi.org/10.1145/3106237.3122821

[42] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1997.

[43] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 685–746, 2001.

[44] K. Ali, X. Lai, Z. Luo, O. Lhoták, J. Dolby, and F. Tip, "A study of call graph construction for jvm-hosted languages," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2644–2666, 2021.

[45] A. Quach, A. Prakash, and L. Yan, "Debloating software through {Piece-Wise} compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 869–886.

[46] C. Porter, G. Mururu, P. Barua, and S. Pande, "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*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 164–180. [Online]. Available: https://doi.org/10.1145/3385412.3386017

[47] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[48] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: Debloating dynamic server-side javascript applications," *CoRR*, vol. abs/2110.14162, 2021. [Online]. Available: https://arxiv.org/abs/2110.14162

[49] R. Wu, H. Zhang, S. Cheung, and S. Kim, "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*, 2014, pp. 204–214. [Online]. Available: https://doi.org/10.1145/2610384.2610386

[50] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.