An Empirical Study of WebAssembly Usage in Node.js

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

Maxwell Bernstein*
acm@bernsteinbear.com
Dartmouth College
Hanover, USA

Daniel Lehmann
mail@dlehmann.eu
Google
Munich, Germany

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

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

Abstract

WebAssembly code executes within a host environment, such as JavaScript running on Node.js. Despite the increasing popularity of mixed JavaScript-WebAssembly applications, the interactions between these two languages and the effect of WebAssembly usage on the NPM ecosystem, are currently not well understood, forcing developers of program analyses and runtime engines and ecosystem tooling authors to make assumptions that may not hold in practice. Moreover, there currently is no executable dataset of WebAssembly modules that allows for studying how WebAssembly is used at runtime within Node.js packages. This paper presents the first comprehensive study of WebAssembly usage in Node.js code. The study is enabled by a novel dataset that we collect comprising 510 executable Node.js packages that exercise 217 unique WebAssembly modules. We study dependencies among packages that use WebAssembly, how JavaScript and WebAssembly interoperate, and the implications for security, efficiency and reliability in the WebAssembly-JavaScript ecosystem. The study provides several insights and future research opportunities, including: (i) a lack of maintenance of WebAssembly binaries that are ports of C/C++/Rust libraries, which motivates future work on cross-language package maintenance, (ii) a lack of testing of WebAssembly usage from JavaScript, which motivates work on targeted testing techniques; (iii) relatively little dynamism in WebAssembly usage, allowing for pragmatic assumptions in program analyses; and (iv) untapped optimization opportunities in engine caching and client-specific debloating. Beyond these insights, we envision our dataset to provide a basis for future studies, program analyses, and work on WebAssembly engine design.

CCS Concepts

• **Software and its engineering** → **Software libraries and repositories**; **Software evolution**; **Maintaining software**; • **General and reference** → **Empirical studies**.

Keywords

WebAssembly, JavaScript, Interoperation, Dependency Analysis, Dynamic Analysis

---

*Work done while at Northeastern University.

1 Introduction

WebAssembly (Wasm) [25] is a low-level bytecode that was introduced in 2017, and was originally designed for computationally-intensive tasks in the browser, e.g., codecs, cryptography, and games. It is widely used in client-side applications and has also seen broad adoption in server-side applications running on Node.js. WebAssembly typically serves as a compilation target from a variety of source languages, including C/C++, Rust, and Go [27].

A WebAssembly binary executes in a *host environment*, which typically consists of JavaScript code. The interactions between a WebAssembly binary and host code can be complex and are currently not well understood. As a result, developers of program analyses, runtime engines and tools are forced to make assumptions that may not hold in practice, which may result in degraded performance and missed opportunities for optimization, as well as unsoundness and imprecision in static analysis techniques. Additionally, the usage of WebAssembly in the NPM ecosystem has not been studied, which can obscure security vulnerabilities, such as buffer overflows, that may arise in WebAssembly binaries [29] when compiled from other languages.

This paper presents the first comprehensive large-scale study of the interactions between JavaScript host code and WebAssembly binaries on the Node.js platform. We study 510 executable NPM packages that collectively exercise 217 unique WebAssembly modules, which we provide as a dataset, named NoWaSet, to the community to facilitate further research. Currently, there exists no dataset of executable WebAssembly modules that interoperate with host code and we envision this dataset to be used by developers of program analyses, JavaScript runtimes and security analyses, as well as authors of ecosystem tooling.

Our study explores quantitatively how host packages interact with and use WebAssembly binaries, to answer questions such as:

- How often, if at all, do host packages update depended-upon WebAssembly binaries that are ports of libraries written in C, C++, Rust, and other languages?
- Which mechanisms (e.g., synchronous, asynchronous, or streaming) are used by host packages to load and instantiate WebAssembly binaries?

- How does host package code interact with WebAssembly binaries by calling functions and accessing elements of WebAssembly binaries, such as function tables?
- How much functionality of WebAssembly binaries is tested by host packages?

Based on our analysis of the observed data, we consider the ramifications for the security, efficiency and reliability in the WebAssembly-JavaScript ecosystem, and identify a number of future research directions that should be explored.

Security: Our study shows that WebAssembly is used pervasively in Node.js via complex NPM package dependencies. Due to WebAssembly's low-level nature and linear memory model, such dependencies may pose security risks [29], particularly when the dependency is indirect. We identify a small number of highly depended upon NPM packages that use WebAssembly, which should be audited carefully to ensure that they do not contain vulnerable or malicious code. We also observe that rather than replacing pre-existing JavaScript libraries, WebAssembly seems to have enabled the reuse of popular systems libraries in Node.js. Packages that depend on WebAssembly binaries that are ports of a C/C++/Rust library, are unlikely to update their binaries to keep up with library updates and so miss out on important security updates. The emergence of WebAssembly has turned NPM/Node into a multi-language ecosystem and package managers have still to catch up with this change.

Efficiency: Our study reveals various optimization opportunities in the JavaScript/WebAssembly ecosystem. For example, we find that WebAssembly modules are often duplicated when an NPM package imports other NPM packages that rely on the same WebAssembly module. Sharing such binaries could remove compilation and instantiation overhead and improve caching of machine code generated by JIT compilers. Our results also indicate that WebAssembly runtimes should cache modules that are instantiated using non-streaming methods, which appear to be favored by Node.js developers. Finally, we also find that packages use some, but not all functionality of a WebAssembly binary, and show that client-specific debloating can be employed to reduce code size.

Reliability: Current static analysis techniques for WebAssembly [1, 12, 13, 38] make unsound assumptions when accounting for interoperation between JavaScript and WebAssembly. Our results show that dynamism is limited in the interoperation, except in the case of function table mutation, suggesting that pragmatic assumptions can be made during program analysis, e.g., assuming that JavaScript does not make calls through the function table. We also find that only a small portion of NPM packages have tests that comprehensively exercise the WebAssembly modules they depend upon which motivates further research into automated testing techniques for multi-language interoperation. Finally, we observe that more than 50% of WebAssembly modules in packages are present as JavaScript Arrays or strings, which is currently not accounted for in JavaScript program analyses.

In summary, this paper makes the following contributions.

- We present the first comprehensive study of interoperation between JavaScript and WebAssembly, providing a quantitative analysis of usage patterns.

- We discuss implications of our findings, providing guidance for work on static analysis, optimization, security risk assessment, and WebAssembly engine design.
- We release NoWaSet, a dataset consisting of 510 NPM packages bundled with 217 WebAssembly binaries that they collectively depend on, to facilitate experimentation on real-world WebAssembly-JavaScript interoperation.
- We discuss research opportunities in the JavaScript/WebAssembly ecosystem, focusing on the development of program analysis, testing and package management techniques in multi-language systems.

The remainder of this paper is organized as follows. Section 2 provides background on WebAssembly. Section 3 covers our methodology. The study results are presented in Section 4. Threats to validity are discussed in Section 5. Section 6 covers related work. Finally, Section 7 presents conclusions and future work.

2 Background

This section presents a short introduction to WebAssembly and how it interacts with host code. For more details, please refer to the website [15], the initial academic publication [25], and the official language specification [39] [40].

2.1 WebAssembly Language Overview

WebAssembly was the first low-level target designed with safe, fast, and portable semantics and an efficient representation. It was designed for performance at par with native code, and for safety through a sandboxed environment. WebAssembly serves as a compilation target for higher-level languages like C/C++ [3, 6], Rust [11] and Go [5] enabling the use of popular libraries on the web.

Figure 1 shows a simple WebAssembly *module*. A module contains a set of *functions* that take in parameters and produce a list of results. A function consists of a sequence of *instructions*, which may load values from, and store values into, a *linear memory*. A module may have a *function table* (W14) which is used for indirect calls. Instruction W09 indirectly calls function bar. Each module has associated *imports* and *exports*. Imports are constructs provided by the environment that is instantiating the module, while exports define constructs that become accessible to the environment after instantiation. *Instantiation* of a module checks that the provided imports match the declared types and initializes the module.

2.2 Interoperation with JavaScript

WebAssembly code is designed to interoperate with a *host environment*. JavaScript is a popular host environment, enabling the use of WebAssembly on the Web and in Node.js [9]. WebAssembly can also be run on standalone runtimes, e.g., Wasmtime [14]. JavaScript and WebAssembly interact, or *interoperate*, via global variables, functions, function tables, and linear memory that are shared via a module's imports and exports. Since JavaScript is a dynamic language, several aspects of its dynamism affect the interoperation with WebAssembly, and in turn, analyses over WebAssembly.

2.2.1 WebAssembly Function Table If a function table is imported or exported, every function in the table becomes reachable from JavaScript. JavaScript can also mutate such a table to add functions

```
J01  const fs = require('fs');
J02  (async () => {
J03    let importObject = {
J04      'host': { 'print': arg => {
J05          console.log(arg);
J06    }}};
J07    let result = await WebAssembly.compile(fs.
           readFileSync("example.wasm"));
J08    let instance = await WebAssembly.instantiate
           (result, importObject);
J09    instance.exports.main();
J10    instance.exports.table.get(0)();
J11  })();
```

```
W01  (module
W02  (import "host" "print"
W03    (func $print (param i32)))
W04  (func $main (export "main")
W05    i32.const 42
W06    call $print)
W07  (func $foo (export "foo")
W08    i32.const 0
W09    call_indirect)
W10  (func $bar
W11    i32.const 23
W12    call $print)
W13  (table $table (export "table") 2 funcref)
W14  (elem $table (i32.const 0) $bar $foo))
```

Figure 1: JavaScript and WebAssembly code that interoperate via calls through imported and exported functions, and via the function table. The orange (and purple) arrows show calls from JavaScript to WebAssembly (and vice versa).

```
J07  LoadAttr,compile | CompileWithType,Uint8Array | DumpCompiledWasm,4df03c.wasm | CompileWithHash,4df03c

J08  LoadAttr,instantiate | InstantiateWithType,WebAssembly.Module | InstantiateWithHash,4df03c | Import,4df03c,host,print,Function

J09  LoadInstanceAttr,4df03c,exports | CallExport,4df03c,main | CallImport,4df03c,host,print

J10  LoadInstanceAttr,4df03c,exports | CallTableExport,4df03c,0 | CallImport,4df03c,host,print
```

Figure 2: Dynamic log generated for JavaScript code in Figure 1.

at run time. Additionally, the offset to the initialization of a table can itself be an imported JavaScript variable, rather than a WebAssembly constant. These aspects of dynamism can have significant repercussions for the development of analyses, such as a static call graph analysis over WebAssembly. If the table is imported or exported, an analysis must consider that every function in the table might be an entry point into the call graph. If the table is initialized with elements starting at a variable offset, an analysis must make worst case assumptions about this offset, which may cause an explosion in the number of edges in the call graph emanating from every indirect call. This introduces significant imprecision, and applications of such a call graph analysis, e.g., dead code elimination or taint tracking, may become much less effective. One of the questions our empirical study aims to answer is whether dynamic practices (e.g., function table manipulation, importing offsets from host code) are common, which may inform static analysis developers whether making conservative assumptions is necessary.

2.2.2 Imported and Exported Functions A static analysis for WebAssembly that makes a closed-world assumption and does not analyze the host code the module interoperates with, is prone to overapproximate the entry points into the module. To be sound, the static analysis must consider every exported function (and every function in an imported or exported function table) to be an entry point into the module. In reality, a client of a C/C++/Rust library compiled to WebAssembly might only use a fraction of the exported functions. One of the goals of our study is to determine to what extent clients of a library compiled to WebAssembly use the entire library and if closed-world assumptions are unnecessarily conservative.

3 Methodology

3.1 Dataset Collection

To the best of our knowledge, there is no previous dataset consisting of executable WebAssembly modules that interoperate with host code. WasmBench [27] is a popular dataset of real-world WebAssembly binaries that is useful for studying static properties of WebAssembly. However, since WasmBench does not preserve host code for binaries that have been scraped from JavaScript packages, it cannot be used to study interoperation.

One could construct a dataset by sampling from the 3 million packages on NPM. During initial experimentation, we began our search process by looking at the top-100 most downloaded NPM packages. However, we only find 2 packages among them that exercise WebAssembly through their tests. Based on this observation, we conclude that WebAssembly is used sparsely in the NPM ecosystem and we adopt a targeted keyword-based approach instead of a random search. A similar approach is taken in related literature: WasmBench [27] finds 3 wasm binaries in the top 1,000 most depended upon NPM packages, versus 1,036 wasm binaries when looking through 2,350 packages that have the keyword 'wasm'. An NPM package can use WebAssembly either directly in its source code or via packages that it depends upon. To find packages that use WebAssembly directly, we search NPM for packages that have the word "wasm" or "WebAssembly" in their keywords (similar to WasmBench[27]). Moreover, to find packages that use WebAssembly indirectly, we search through the dependents of these packages. Our search process is illustrated in Figure 3, which essentially is a breadth-first search through these packages and their dependents. The following describes the process in detail.

We search through 2,008 packages scraped from NPM for *viable* packages that we add to the dataset. A package p is viable if: (i) p builds successfully, (ii) p has passing tests, and, (iii) at least one of p's tests executes WebAssembly. We use `npm-filter` [17] in our filtering process. If a package installs and builds successfully on NodeJS v21.7.1 and has successfully running tests, we check that the package contains a `.wasm` binary or that the word `WebAssembly` appears in any JavaScript file. If a package passes this check, we run its passing tests on a version of node with support for WebAssembly

disabled. If this causes a previously passing test to fail, we determine that the test exercises WebAssembly. If p passes the above checks, we add it to the dataset and scrape GitHub for its dependents using the `ghtopdep` tool [23]. We sort through the dependents of package p by the number of stars and analyze the top-30 dependents of p, with a minimum of 5 stars, to see if they are viable. We impose a minimum of 5 stars to filter out template repositories that have no tests, and only add the top-30 dependents so as to not have an over-representation of the dependents of popular packages.

As seen in Figure 3, the number of packages to be analyzed increases with increasing depth. Examining a package takes an average of 10 minutes. Due to computational constraints, we terminate the dataset collection process when 5,000 unique packages have been examined. This results in NoWaSet, a dataset consisting of 510 packages that use WebAssembly. Note that the overall incidence of WebAssembly is low even during a directed search through the search space. Only 10% of packages are found to depend on WebAssembly and exercise it through tests. A completely random search would find an even lower incidence of WebAssembly, and it would be far more computationally expensive to collect a dataset of the same size through random sampling. Figure 3 presents several metrics for the dataset, which we calculate using the analyses described in Section 3.4 and 3.5. Our static results include an additional 54 packages whose builds failed due to linter errors from instrumentation in the dynamic analysis.

3.2 Research Questions

The main goal of our study is to determine how JavaScript developers use WebAssembly in practice. In this section, we refine this goal into five research questions.

RQ1: How do Node.js packages depend on WebAssembly? To study the usage of WebAssembly by clients in the dataset, we look at how NPM packages depend on WebAssembly modules. WebAssembly might be present in a package's source code or in its dependents. Since previous work has not studied the prevalence of WebAssembly in NPM packages, JavaScript developers and program analysis writers might not be accounting for its presence in their packages through dependencies. Additionally, security vulnerabilities, e.g., buffer overflows [29] in WebAssembly modules, might go unnoticed and affect a large number of clients, if exploited.

RQ2: How has WebAssembly usage in NPM packages evolved over time? WebAssembly was originally introduced with the goal of being able to perform computationally intensive tasks on the Web safely and to allow for reuse of popular C/C++/Rust libraries on the Web [25]. To study if this goal has been met, we study the evolution of WebAssembly usage in the NPM ecosystem by looking at when and why packages either introduce WebAssembly or switch to using WebAssembly from JavaScript. We also study whether packages upgrade to new versions of Wasm binaries that they depend on.

RQ3: How comprehensively do client packages in the dataset test the WebAssembly modules they depend upon? Node.js packages that depend on WebAssembly modules should test the interaction with these modules to ensure that they work as expected, offer the desired performance, and that any errors at the interface

between the two languages are handled properly. We use three metrics to measure the comprehensiveness of a package's tests: (i) the number of WebAssembly modules dynamically instantiated by tests, (ii) the number of WebAssembly modules that are interoperated with, and (iii) the variance exhibited in the interoperation.

A package can depend on several WebAssembly modules through its dependents. Analyzing the number of WebAssembly modules that are dynamically instantiated during test execution and comparing it to the number of WebAssembly modules that can be statically determined to exist in a package, gives us a meaningful estimate of how well WebAssembly modules are tested. Additionally, a test might instantiate a module without interoperating with it, that is, without calling any of its exported elements. Instantiating a module does not suffice to test the module, so we also measure how often a client to a WebAssembly module interoperates with it. We record the variance in interoperation to assess if clients use a WebAssembly module in different ways or in the same way.

RQ4: How are JavaScript program analysis and engine developers affected by the presence of WebAssembly? WebAssembly has seen broad adoption [27] and program analyses and engine developers have to reckon with its presence. This necessitates thinking about how WebAssembly is distributed and how interoperation between JavaScript and WebAssembly affects the analysis of both languages.

The WebAssembly JavaScript API [16] presumes that WebAssembly modules are most commonly instantiated by having web browsers making HTTP requests for a `.wasm` binary file. However, WebAssembly can also be inlined in JavaScript by encoding the raw bytes of the module as a base64-encoded string or in an `ArrayBuffer`[1]. We report on the prevalence of WebAssembly as binary files versus inlined modules.

The WebAssembly JavaScript API provides three possible ways to instantiate a module [7] (i) an asynchronous API that takes the raw bytes for a WebAssembly module and an import object that returns an instantiated module, (ii) a streaming API variant of (i), and (iii) a synchronous method that takes in a compiled module and returns an instantiated module. The documentation suggests that the asynchronous streaming method is preferred and some JavaScript engines, such as V8, only cache WebAssembly modules instantiated using the streaming method [22]. We measure the prevalence of these instantiation methods in practice.

Prior work [31] reports that current state-of-the-art static analysis tools [13] [1] [12] [38] over WebAssembly binaries make a closed-world assumption about entry points into a WebAssembly binary. They assume that one can analyze WebAssembly in isolation without considering its interoperation with a host language. While this is a reasonable assumption to make in the case of standalone execution of WebAssembly, it foregoes all WebAssembly that interoperates with JavaScript. The same prior work also recognizes that the dynamism of JavaScript leads to high imprecision while analyzing WebAssembly binaries. For example, JavaScript can mutate an imported or exported function table and call functions through this table, both of which lead to imprecision in a WebAssembly call graph. Similar to previous work that studies the dynamism

---

[1]Compilers like Emscripten have optimization flags [10] that inline WebAssembly in JavaScript. This is done, e.g., to avoid the overhead of an additional HTTP request.

$$\text{viable package } p := \begin{cases} p \text{ builds successfully } \wedge \\ p \text{ has passing tests } \wedge \\ \text{at least one test fails when} \\ \text{WebAssembly support is disabled} \end{cases}$$

METRICS FOR NOWASET	
NPM packages that use WebAssembly	510
○ Packages that directly instantiate WebAssembly	27
○ Packages that indirectly instantiate WebAssembly	483
Statically detected WebAssembly modules	1,257
Instantiated WebAssembly modules	217
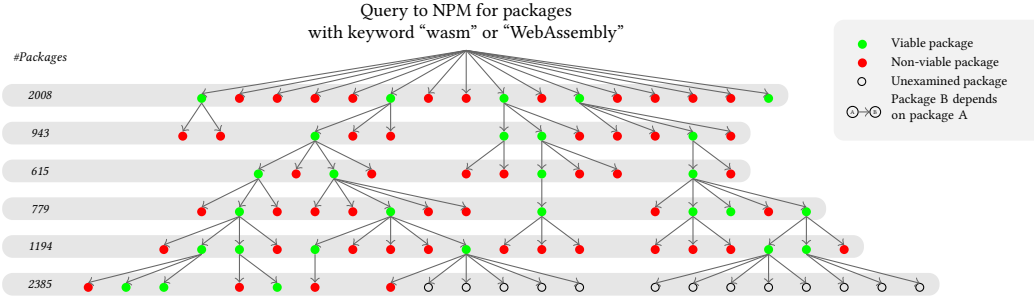
Figure 3: Breadth-first search through package dependency tree to find viable packages for the dataset.

exhibited by JavaScript code [36], we report on the dynamism exhibited in JavaScript/WebAssembly interoperation to determine if developers can make pragmatic assumptions in their analyses.

RQ5: What optimization opportunities exist for client packages that use WebAssembly? JavaScript packages often depend on hundreds of other packages. In RQ1, we study how packages depend on WebAssembly. If packages mostly depend on WebAssembly indirectly and there are WebAssembly modules that are highly depended upon, there might exist duplication of WebAssembly modules within a package through these indirect dependencies. We report on the number of WebAssembly module duplicates present in packages so that JavaScript engine developers can account for it while caching bytes of the WebAssembly module.

All state-of-the-art static analysis tools over WebAssembly binaries assume every function exported from WebAssembly to be reachable from JavaScript. As shown by Lehmann et al. [31], this assumption leads to high imprecision, and it is possible that JavaScript clients rarely call all functions exported by a module. If modules are commonly instantiated but never interoperated with, JavaScript engines can opt for lazy compilation of modules. If there is large variation in the percentage of exported functions called by clients, client-specific debloating could help reduce code size.

3.3 Tracking the Evolution of WebAssembly in NPM packages

To study how the usage of WebAssembly in packages has evolved (RQ2), we manually study a randomly selected pool of 50 packages from the dataset that use WebAssembly in their source code, rather than in their dependencies. We restrict ourselves to a subset and do not analyze all 510 packages in the dataset because it is very difficult to programmatically determine the source of a WebAssembly binary, in order to track its evolution or updates to the original library. Often a package developer has manually built a wasm binary from C/Rust library code and included it in their dependencies, but do not indicate in source code which library this binary was built from.

For each package, we study their commit history and note how WebAssembly usage was introduced in the application, that is, if the package switched to using WebAssembly from JavaScript or if the WebAssembly binary is a port of a library in a different language (C/C++/Rust/Go). We note down the date of the commit that introduced the WebAssembly binary. If the WebAssembly binary is a port of a library, we manually analyze commit messages, JavaScript wrapper files and wasm binaries to determine their source library. We also track if, and how often, the binary was updated to keep up-to-date with the original library updates. This is done to study if and how the depended upon WebAssembly binary is maintained.

3.4 Statically Detecting WebAssembly Modules

To determine the most depended upon NPM packages that use WebAssembly (RQ1), study the comprehensiveness of package tests (RQ3), and report on the prevalence of inlined WebAssembly versus WebAssembly binary files in packages (RQ4), we run a static analysis to detect WebAssembly binaries and the inlined WebAssembly modules in an NPM package. All WebAssembly modules start with a header containing the byte string \0asm, also called the Wasm binary magic number. A JavaScript array or string contains the raw bytes of a WebAssembly Modules if it starts with this magic number. We parse every JavaScript file in a package's source code and in the source code of all its dependents to determine the prevalence of inlined WebAssembly modules in the package. We record the hash of the raw bytes of every inlined WebAssembly module and .wasm file, to be able to de-duplicate the different WebAssembly modules.

For each package in the dataset, we generate a dependency tree for its dependent packages that contain WebAssembly modules. We analyze the path to a file that contains one or more WebAssembly modules and extract the package name as well as the dependencies of this package. We record the package dependency tree as well as all files in the package that contain WebAssembly modules.
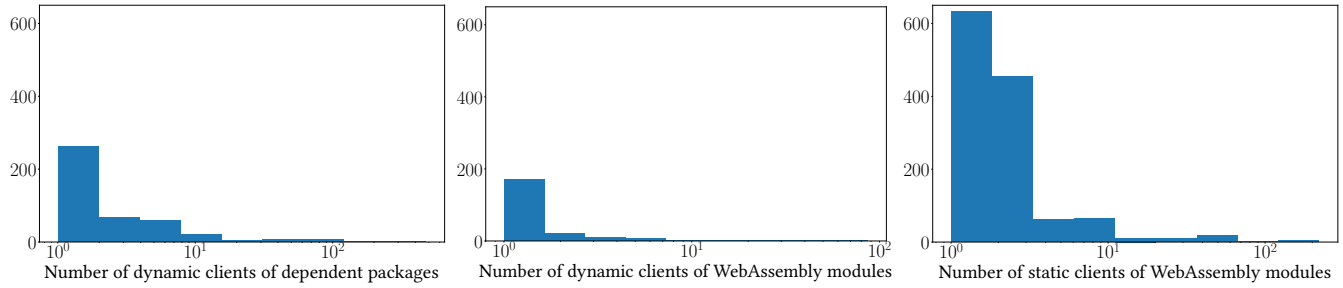
Figure 4: Frequency distribution over the number of static and dynamic clients of dependent packages and WebAssembly modules.

3.5 Dynamically Analyzing Interoperation

To answer questions related to package tests, instantiation of Web-Assembly modules, and JavaScript/WebAssembly interoperation (RQ3 to RQ5), we run a dynamic analysis that injects logging code into each JavaScript file and captures logs over test execution of each package, as shown in Figure 2. For each dependent package of a NoWaSet package that contains WebAssembly, we instrument the package and run tests to collect dynamic logs associated with the package. We also collect logs associated with the package itself by only instrumenting its source code.

The injected logging code instruments all methods that could be used for WebAssembly module compilation and instantiation, as well as calls to exported functions and the `set` and `get` methods of the function table. For each of these methods, we hash the WebAssembly module contents and write the modules bytes and hash to a file so that we can reference it later when analyzing the logs. We associate each dynamic log with its module hash, so as to differentiate between multiple WebAssembly modules that might be instantiated within the same test execution. In Figure 2, we see that a call to `WebAssembly.compile` (J07) generates a log that states that the compile method was called on a `Uint8Array`, to create a module with the hash `4df03c`. J08 generates a log that states that a compiled module with hash `4df03c`, is being instantiated with the JavaScript `print` function. The dynamic log for J09 shows that after the exported `main` function is called by JavaScript, the imported `print` function is called. The dynamic log for J10 also illustrates that a call to a function in the exported function table has been logged.

4 Empirical Results

We answer the research questions outlined in Section 3.2 by collecting static data and dynamic logs over the 510 packages in the NoWaSet dataset. We refer to these packages as client packages of the WebAssembly modules. A package may depend on several other NPM packages, each of which might contain WebAssembly modules.

4.1 How do Node.js packages depend on WebAssembly?

We study how WebAssembly modules are depended upon by packages in the dataset. This helps us understand the prevalence of WebAssembly in the dataset and if there are popular NPM packages and WebAssembly modules that a large portion of the dataset depends upon.

4.1.1 Indirect Use of WebAssembly 510 packages in the NoWaSet dataset depend on WebAssembly directly, through their own source code, or indirectly, through dependent NPM packages. 27 packages only depend on WebAssembly directly while, 313 packages only depend on WebAssembly indirectly. We observe that some of these packages are source code repositories for NPM packages that others are clients of. 62.59% of NoWaSet packages depend on more than one NPM package with WebAssembly and 75.71% of them ship with more than one WebAssembly module. This means that if a package contains WebAssembly, it is likely to depend upon different packages that use WebAssembly, who in turn are likely to have more than one unique WebAssembly module. This suggests that there exist complex dependencies among packages that use WebAssembly, which poses challenges for program analysis developers.

> **Insight.** WebAssembly is used in the Node.js ecosystem via complex NPM package dependencies, which suggests the presence of WebAssembly has to be accounted for by JavaScript program analysis developers.

These complex dependencies suggest that there might be packages that are commonly depended upon. Prior work [29] has discussed how security vulnerabilities, e.g., buffer overflows, can show up in WebAssembly modules from their source code languages. These vulnerabilities could affect the clients of popular packages that use WebAssembly. Dependencies between packages and their security implications have been discussed in other studies [42]. We extend this line of work to NPM packages that use WebAssembly directly and report the most depended-upon packages that use WebAssembly.

(1) `source-map` is a package used to debug minified JavaScript, on which 183 packages depend.
(2) `cjs-module-lexer` is a package that detects the most likely list of named exports of a CommonJS module, on which 135 packages depend.
(3) `es-module-lexer` is similar to the above package. 134 packages depend on it.
(4) `@webassemblyjs/helper-wasm-bytecode` is a package that consists of various utility tools over WebAssembly binaries. 117 packages depend on it.
(5) `@xtuc/long` is used to construct 64 bit two's-complement integers and perform operations on them. 116 packages depend on this package.

Insight. Some NPM packages and WebAssembly modules are highly depended upon and should be audited to make sure they do not contain malicious code and vulnerabilities.

4.2 How has WebAssembly usage in NPM packages evolved over time?

We manually study the evolution of WebAssembly usage in the NPM ecosystem for a randomly selected subset of 50 packages that depend on WebAssembly directly. The introduction of Web-Assembly in NPM packages has increased steadily over time, with 30% (15/50) packages starting to use WebAssembly in the past two years. The WebAssembly in packages is more often than not a port of a library in a different language like C/C++/Rust. 34% (17/50) of packages exhibited this pattern while 10% (5/50) of packages switched from using a JavaScript library to using WebAssembly. Of these packages, all cited performance improvements as a reason for the switch. For packages that switched from using JavaScript to using WebAssembly (Eg: source-map), we investigated clients of these packages to track if the change had been adopted widely. We found that switching to WebAssembly caused the API of these packages to change to an async model. This is an invasive change, which only 50% of clients adopted and the others choose instead to stick with the version of the library that used JavaScript instead of WebAssembly.

Insight. Rather than replacing pre-existing JavaScript libraries, WebAssembly seems to have enabled the reuse of popular systems libraries on the web.

We could match up the WebAssembly depended upon by a package with a certain library for 41 packages. Of these, 66% (27/41) packages do not update the WebAssembly binaries to keep up to date with library updates. Of the remaining 14 packages that update their WebAssembly binaries at least once, the WebAssembly binary is updated an average of 5.7 times, which the corresponding library has 15.5 releases or updates. This suggests that the package management problem that C libraries face is bleeding into the management of WebAssembly binaries. This especially does not bode well considering that some of the updates are important security updates. We plot the updates to WebAssembly binaries depended upon by packages and updates to their original libraries in Figure 5 for two packages, while also showing security updates to these libraries. We study the library's commit messages and release notes to determine if a release contains a security update.

Insight. Packages that depend on WebAssembly binaries that are ports of a C/C++/Rust library, are unlikely to update their binaries to keep up with library updates.

4.3 How comprehensively do packages in the dataset test the WebAssembly modules they depend upon?

In Section 3.2, we discuss the importance of testing WebAssembly modules in a package. We first study how often packages dynamically test the WebAssembly modules in their source code and in their dependent packages. We also delve into how often these packages interoperate with WebAssembly modules and if they exhibit variance in their interoperation.

4.3.1 Dynamic Clients in NoWaSet Analyzing how often a NoWaSet package dynamically exercises the packages and modules it depends upon helps us determine how often these WebAssembly modules are interacted with. We analyze the number of static clients and dynamic clients, determined through client test execution, of various NPM packages in the dataset and report their frequency distributions in Figure 4. 70.4% of NPM packages in the dataset that have been statically determined to have WebAssembly, have upto three dynamic clients. Meanwhile, 83.69% of WebAssembly modules in the dataset have up to three static clients but 78% of modules have only one dynamic client. The large discrepancy between the number of static and dynamic clients of WebAssembly modules tells us that while NoWaSet package tests might exercise their dependent NPM packages, they do not manage to exercise the modules in these packages comprehensively.

4.3.2 Interoperation with WebAssembly modules 217 unique Web-Assembly modules are dynamically instantiated by package tests. However, only 110 (50.5%) of these show interoperation with Java-Script. We consider a module to be interoperated with if a test calls an exported function of the module or calls a function in an exported function table. This suggests that many WebAssembly modules are instantiated and not interoperated with. Through manual inspection of these packages, we find that these WebAssembly modules are often C/C++ or Rust libraries that have been compiled to WebAssembly and shipped as part of a NPM package. The NPM package often contains some setup code that instantiates the Web-Assembly module, but then never tests it. While it may be safe to assume that the original library has been tested, developers of Node.js packages should test the interaction between their Java-Script code and the compiled WebAssembly module. For example, such tests could check that data is correctly passed between Java-Script and WebAssembly, that the WebAssembly module offers the expected performance properties, and that the JavaScript code correctly handles the output, including any errors, of the WebAssembly module. Having a comprehensive test suite for this interaction is also important for maintaining the package in the future, as it can help ensure that changes to the WebAssembly module do not break the JavaScript code that depends on it.

4.3.3 Variance in Interoperation with JavaScript 107 WebAssembly modules are called from JavaScript via exported functions. Of these, 79 modules have one client and 28 are called from from more than one client. 18 modules have its clients call the same set of exported functions while 10 have different calls from clients. Among modules whose clients call the same set of exported functions, we observe that this is usually a setup function that is called by the clients, and no other exported functions are called. This suggests low coverage of WebAssembly modules' exported functions.

Insight. Only a small portion of packages comprehensively test the WebAssembly modules they depend upon. Future work could develop automated testing techniques to further test WebAssembly in NPM packages.
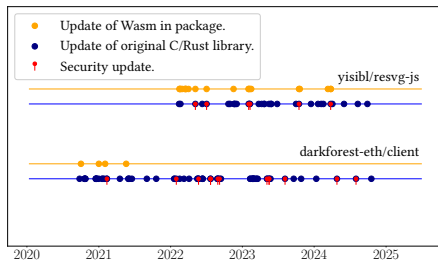
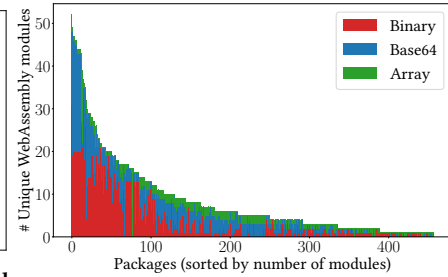**Figure 5: Frequency of updates to depended upon Wasm binaries vs their source libraries.**

Figure 6: Prevalence of methods of distributing WebAssembly.
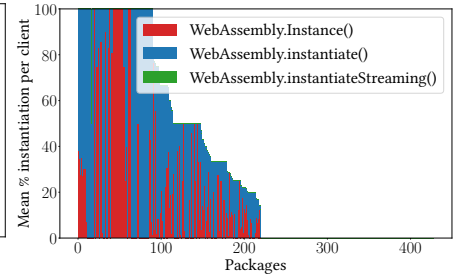

Figure 7: Instantiation type of WebAssembly modules in packages.

In the remainder of the study, we draw conclusions based on packages that exercise WebAssembly modules through tests. While there is a need for further testing of WebAssembly in packages, we rely on existing tests to show us how JavaScript and WebAssembly interoperate. Since we have a large dataset, we have dynamic data for 217 unique WebAssembly modules.

4.4 How are JavaScript program analysis and engine developers affected by the presence of WebAssembly?

The pervasive presence of WebAssembly has to be accounted for in JavaScript by both program analysis developers and engine developers. To inform various design decisions for these analyses, we study how WebAssembly is distributed, how it is instantiated, and the dynamism exhibited by the interoperation between the two languages.

4.4.1 Different Methods of Distributing WebAssembly Our analysis over different methods of distributing WebAssembly modules in the NoWaSet dataset finds that all three methods are prevalent, i.e., WebAssembly is found to exist as binary files, as base64-encoded strings and in array expressions. Figure 6 visualizes the prevalence of various distribution methods of WebAssembly modules in the studied packages. The figure excludes four packages from the dataset that are outliers. Three of the four outliers have over 400 WebAssembly binary files that serve as language specification tests. The fourth outlier is a JavaScript-polyfill for hyphenation in HTML, which has 300 WebAssembly binary files to encode various hyphenation patterns. Packages with more than 45 unique modules depend on an package called `hash-wasm` that uses hand-tuned WebAssembly modules encoded in minified JavaScript to calculate hashes.

We find that binaries are the most popular source of WebAssembly modules in the dataset, but only by a small margin:

(1) 48% of modules are encoded as `.wasm` binary files.
(2) 39% of modules are inlined as base64-encoded strings.
(3) 13% of modules are inlined as arrays.

This would suggest that more than half of developers using WebAssembly use it inlined within JavaScript. Manual inspection of packages in the dataset with inlined modules leads us to hypothesize that developers prefer this method of distributing their WebAssembly modules due to faster performance. Inlined WebAssembly does not need to be fetched from disk.

The distribution mechanism of WebAssembly modules is important for developers of engines and program analyses. For client-side packages, network caching performed by the browser, that is, caching the bytes of the WebAssembly module itself, should be independent of the method of distributing WebAssembly. Additionally, current state-of-the-art static analysis tools are built to only analyze WebAssembly binary files and instead should integrate a host analysis to be able to analyze inlined WebAssembly files as well.

In addition to WebAssembly modules being instantiated by client packages, we find 96 modules that are used internally by the Node.js engine. For example, Node.js uses the `undici` package for making performant HTTP requests, and this package in turn uses the `llhttp` package, which uses WebAssembly. Another example is that the Node.js engine uses WebAssembly to detect the named exports of a CommonJS module. We omit these packages when discussing methods of distributing WebAssembly in NoWaSet packages.

> **Insight.** The presence of inlined WebAssembly in JavaScript files is significant and JavaScript program analysis, engine developers should account for it by identifying WebAssembly binaries using the wasm binary magic number.

4.4.2 Instantiation of WebAssembly modules A WebAssembly module can be instantiated in three ways: synchronously via `WebAssembly.Instance`; asynchronously via `WebAssembly.instantiate`, which returns a promise that resolves once instantiation has finished; or in a streaming manner via `WebAssembly.instantiateStreaming`, which takes as input a fetch request of the module. Of these, the streaming method is the most performant, since it can validate and compile code while the download is still in progress.

Figure 7 shows the percent average instantiation per client for the 510 packages in the dataset. Note that clients of a package may or may not instantiate WebAssembly modules in the package. We see that over half of the NPM packages in the dataset do not have clients that dynamically instantiate the WebAssembly modules they contain, which is to be expected considering the results discussed in Section 4.3. 39.5% of instantiated modules are instantiated through the synchronous instantiation method, 60.15% are instantiated using the asynchronous non-streaming method and less than 0.5% of modules are instantiated using the asynchronous streaming method. This is because streaming is only important for packages that run on the web and fetch WebAssembly through relatively slow network requests. Since we inspect packages where WebAssembly is loaded
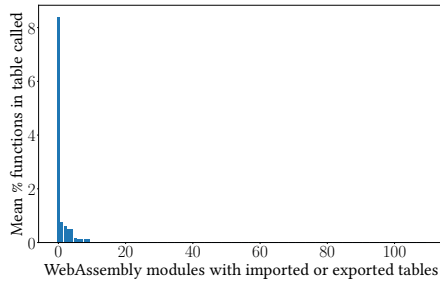
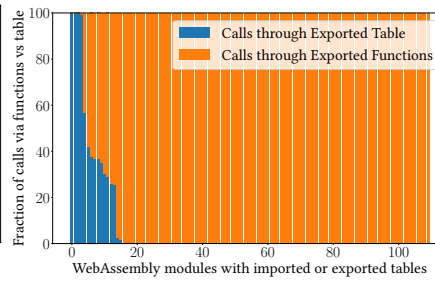**Figure 8: Mean percentage of functions called by JavaScript through exported tables.**

Figure 9: Fraction of calls from JavaScript through functions vs function tables.
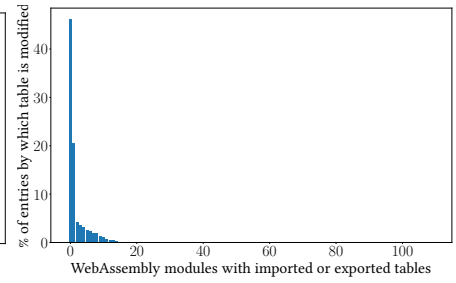

Figure 10: Percentage of the original entries by which an exported table is modified.

from disk or via inlined bytes, there is less need for streaming, which explains the low prevalence of calls to the streaming method. However, front-end packages still operate on the web and should use the streaming instantiation method. Additionally, engines like V8 [22] only cache generated code for modules that are instantiated through the streaming method, which means that generated code for very few of the modules being instantiated is being cached.

> **Insight.** Engines might consider caching WebAssembly that is instantiated using non-streaming methods, since those are the predominant methods used by JavaScript packages in the wild.

4.4.3 Dynamism Exhibited in Interoperation JavaScript is a highly dynamic language whose interoperation with WebAssembly can also be very dynamic [31]. Generally, more dynamic behavior leads to program analysis developers having to make assumptions about program behavior, which negatively affects the precision or the completeness of the analysis. We study the dynamism exhibited by JavaScript in its interoperation with WebAssembly to see its effect on program analysis development.

First, we look at the offset to the initialization of a WebAssembly table, which can be a value read from JavaScript at the time of instantiation. Of the 217 unique WebAssembly modules that are dynamically instantiated, all contain a function table. Of these, 145 have element sections whose offsets are initialized with a static WebAssembly constant and only two are initialized with an imported variable. Both of the imported offsets are set to zero at runtime. These results suggest that analysis developers can make the pragmatic assumption that the offset variable is always zero.

Next, we look at how many functions are invoked by JavaScript through a WebAssembly function table, rather than exported functions. Out of 110 WebAssembly modules that export a function table, only 10 have functions called through it. Figure 8 shows the percentage of functions in the table that are actually called. The percentage varies between 0% and 8.5%, with a median of 0.33%. JavaScript could also call functions through an imported WebAssembly function table, but it never does. In Figure 9, we compare the percentage of calls made by JavaScript via exported functions versus through an exported table and can see that only 5 modules have more than 50% of their calls through a table.

We also look at how many packages mutate an imported or exported WebAssembly table. 16 of the 110 WebAssembly modules with exported tables, have their table modified by JavaScript. Figure 10 shows that only two modules have more than 5% of their

entries modified. 14 modules have their table modified between 0% and 5% of its original size, with an average of 1.75%. We conclude that the mutation of function tables by JavaScript code is non-negligible and should be accounted for by analysis developers.

> **Insight.** Program Analysis developers can make the pragmatic assumption that the dynamism exhibited between JavaScript and WebAssembly is limited, except in the case of table mutation.

4.5 What optimization opportunities exist for packages that use WebAssembly?

4.5.1 Duplicates within a package Large WebAssembly binaries are expensive to compile. Browers cache compiled WebAssembly module data using IndexedDB [2], an API for client-side storage, however no such alternative exists for Node.js [8]. Prior work [33] suggests that such a shared code cache would improve load times. We report on the number of duplicate WebAssembly modules found in each package in the dataset, in order to quantify the potential cost of re-compilation. An average of 26.57% and a median of 30% of modules in a package are duplicated due to multiple dependent packages depending on the same WebAssembly module.

> **Insight.** WebAssembly modules are often duplicated within a package. Node.js should employ a compiled module cache to reduce the cost of re-compilation.

4.5.2 Exported Functions Called by JavaScript 217 unique WebAssembly modules are dynamically instantiated by package tests. However, only 50.5% of them are also interoperated with through JavaScript, that is, JavaScript calls an exported function or a function through an exported table. If only small subsets of a WebAssembly module are actually executed, compiling it on first execution would be beneficial. Some engines, e.g., V8, employ such a lazy compilation strategy, which should be extended to other engines as well.

For the 110 unique WebAssembly modules that are interoperated with, Figure 12 reports on the average percentage of exported functions dynamically called. We see that 43% of modules have between 5% and 60% of their exported functions called. A WebAssembly module in the dataset has a mean of 61.4% and median of 60.8% of its exported functions called by a client. This data suggests that clients of a package are only making use of a portion of the WebAssembly exported functions. To observe the correlation between the number of exported functions in a WebAssembly module and the
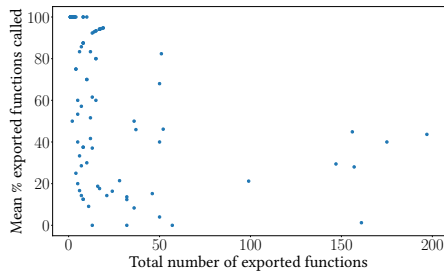
Figure 11: Scatter plot of total functions vs those called by JavaScript.
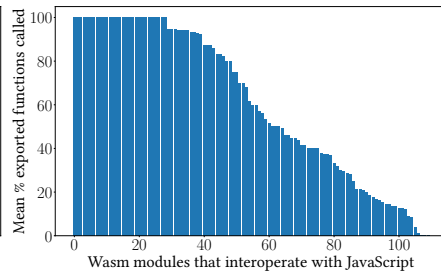
Figure 12: Mean percentage of functions called by JavaScript.
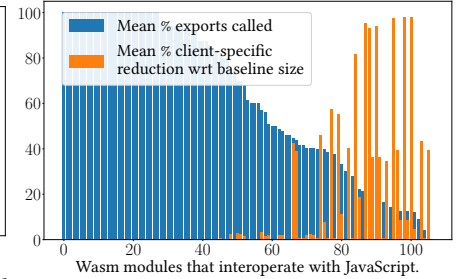
Figure 13: Mean percentage size reduction of binaries using MetaDCE for debloating.

percentage of exports called by a client, we show a scatter plot of the two values in Figure 11. For better readability, the figure excludes WebAssembly modules with more than 200 exported functions, which accounts for 9% of all WebAssembly modules. We observe that, as the number of exported functions increases, the percentage of functions called by JavaScript falls. This further suggests partial use of modules by clients.

In order to study the potential of client-specific debloating for WebAssembly binaries, we run a state of the art dead-code elimination tool from the Binaryen toolchain [1], MetaDCE (version 119), on the binaries in the dataset. Previous work [31] has studied MetaDCE and other state of the art WebAssembly static analysis tools, showing that they all make unsound assumptions about client behavior. Most of these tools do not allow for the input of client-specific information and are rarely able to identify dead code. However, MetaDCE allows the input of client-specific information in the form of a reachability graph that indicates which exported functions are really used. We generate this reachability graph from dynamic logs that report on the exported functions that are called during test execution. Figure 13 reports on the percentage size reduction of the binaries after dead-code elimination when compared to a baseline of dead-code elimination where all exported functions are made reachable from the client. We can see that including client-specific information about the exports being called, means that MetaDCE is able to specialize the binary to a client. MetaDCE is not perfect – its call-graph analysis is potentially unsound when determining the targets of an indirect call [31]. Given the potential for reducing the size binaries we find, future work should investigate how to improve the soundness and precision of MetaDCE and related tools, and to employ them more frequently to specialize WebAssembly binaries to a client.

> **Insight.** A client-specific debloating strategy can specialize WebAssembly binaries to their clients and reduce code size of packages.

5 Threats to Validity

As for any empirical study, the conclusions drawn from our dataset may not be generalizable to all JavaScript packages and WebAssembly modules. For example, our suggestions for pragmatic assumptions in program analyses are not guaranteed to hold for all WebAssembly modules. Our methodology is designed with the goal of creating a dataset of executable WebAssembly code, but not meant to cover a representative sample of all JavaScript packages. This goal is reflected in our choice to target Node.js, but not WebAssembly in the browser. We refer to work on recording and replaying WebAssembly in the browser [19] for a complementary approach to gathering an executable dataset. Additionally, our dataset search process biases our findings toward WebAssembly being used by developers or in programmatic and debugging contexts. This is particularly evident in the list of packages that are highly depended upon in our dataset (RQ1). Since the original use case for WebAssembly is better performance and re-use of old libraries, the high usage of WebAssembly in developer oriented packages is unsurprising. Finally, our results are constrained by the quality and comprehensiveness of the test suites of the repositories in the dataset.

6 Related Work

WebAssembly. The initial version of WebAssembly [39] (often dubbed MVP or 1.0) was standardized in 2015, with several proposals having been merged into the standard since [40]. WebAssembly's interoperation with a host environment is currently being formalized under the Wasm Component Model Proposal [4].

Other WebAssembly datasets and studies. Musch et al. [32] performed the first large-scale study of WebAssembly usage on the web, using the Alexa Top 1 million websites ranking. Similar to our dynamic analysis, they hook the creation of all JavaScript functions that compile or instantiate WebAssembly modules in a Google Chrome crawler and visit each website. They study the extent of usage of WebAssembly on these websites and inspect different applications of WebAssembly. They do not inspect WebAssembly usage in Node.js, its interoperation with JavaScript, and do not provide an executable dataset from their data collection. WasmBench [27] is a dataset of 8,461 real-world WebAssembly binaries from GitHub, package managers and the Web. WasmBench is not an executable dataset and so can only be used to study static properties of real-world binaries. [27] also study how WebAssembly is compiled to, vulnerabilities in WebAssembly originating from different source languages and different use cases of WebAssembly. Wasm-R3 [19] is a record-replay approach to gathering WebAssembly in the browser to produce a standalone WebAssembly benchmark. JABBERWOCK [28] is a tool for WebAssembly dataset generation that works by transforming JavaScript in existing websites to WebAssembly code, which is then used to detect malicious websites using Machine Learning. Lehmann et al. [31] study the problem of

static call-graph construction over WebAssembly binaries and provide a set of microbenchmarks to reflect the identified challenges. They also provide a small set of real-world WebAssembly libraries with executable tests.

Program analysis of WebAssembly. Wassail [38] is a research static analysis toolkit for WebAssembly binaries that includes call graph analysis, taint tracking and slicing. MetaDCE [1] is an optimizer from the Binaryen tool suite that performs dead code elimination. Twiggy [12] is a code size profiler for WebAssembly binaries, that detects dead code. Wasmati [20] is a static vulnerability scanner for WebAssembly binaries. Wasabi [30] is a general framework for implementing dynamic analyses over WebAssembly binaries. WAFL [26] is a fuzzer for WebAssembly binaries. [33] studies compilation performance and shared code caching in Node.js for WebAssembly modules over the PolyBench/C dataset.

Studies of dynamism in other languages. Astrauskas et al. [18] study how programmers use the Unsafe construct in the Rust programming language by querying a large database of publicly available Rust code. Yang et al. [41] statically study complex Python features to determine the usage of dynamic features, and other work [34] studies how developers use type annotations and type checking and inference tools in Python. Goel et al.[24] analyze 49 million calls to a construct in the R programming language called eval to understand why programmers use eval. Others [35] did similar work studying how a similar construct in JavaScript is used by programmers by collecting and analyzing dynamic traces.

Other multi-language studies Other studies in the multi-language setting follow a similar methodology to us and have similar aims and outcomes, like offering pragamatic insights to tool writers. Rigger et al. [37] analyze the presence of x86-64 inline Assembly in C programs and offer practical insights like the most common x86 instructions that tool writers can support to support a majority of C projects with inlined assembly. Such a insight is less relevant for inlined WebAssembly —x86 has roughly 1500 instructions in its instruction set while WebAssembly has less than 300. Chaliasos et al. [21] analyze inlined assembly in Solidity Smart Contracts and draw similar conclusions to us with regards to why assembly is used in a multilanguage setting, namely, to implement functionality that isn't available in a host language and that program analysis developers do not account for its presence. We do however draw insights specific to the WebAssembly domain: that inlining is done due to the network latency constraints in a JavaScript browser context where, depending on the file size, the overhead of a HTTP request can be optimized away through inlining.

7 Conclusions and Future Work

We have presented the first comprehensive study of WebAssembly usage in Node.js packages. In this study, we investigate how often host packages update WebAssembly binaries they depend upon, mechanisms used by host packages to load and instantiate WebAssembly binaries, how host code interacts with WebAssembly by calling functions and accessing function tables, and how much functionality of WebAssembly binaries is tested. A large part of our study is enabled by the construction of NoWaSet, a novel dataset of 510 executable NPM packages that collectively exercise 217 unique

WebAssembly modules, which we make available to the community to facilitate further research.

The study provides several insights. This includes the existence of a small number of highly depended upon NPM packages that rely on WebAssembly modules, which should be carefully audited to ensure they contain neither malicious nor vulnerable code. We also observed a lack of testing of WebAssembly usage from JavaScript, which motivates work on targeted testing techniques. We found relatively little dynamism in WebAssembly usage, enabling for pragmatic assumptions to be made in program analyses. Moreover, the study identified several untapped optimization opportunities, such as the presence of duplicate WebAssembly modules and situations where a host packages uses only a subset of a WebAssembly module's function, which suggests that client-specific debloating techniques may be useful.

As future work, we plan to develop static analysis techniques that are capable of precisely analyzing packages that rely on WebAssembly binaries, by leveraging the insights gained from conducting the study presented in this paper.

8 Data-Availability Statement

The NoWaSet dataset, the scripts used in dataset collection, the program analyses used in the study as well as detailed logs from every test and scripts to interpret these logs into the results and graphs in the paper are all available at https://github.com/michelledaviest/NodeWasmStudy.

9 Acknowledgements

References

[1] 2024. *Binaryen.* https://github.com/WebAssembly/binaryen
[2] 2024. *Caching compiled WebAssembly modules.* https://udn.realityripple.com/docs/WebAssembly/Caching_modules
[3] 2024. *Clang: a C language family frontend for LLVM.* https://clang.llvm.org/
[4] 2024. *Component Model design and specification.* https://github.com/WebAssembly/component-model
[5] 2024. *Introduction to WebAssembly using Go.* https://golangbot.com/webassembly-using-go/
[6] 2024. *The LLVM Compiler Infrastructure.* https://llvm.org/
[7] 2024. *MDN Docs: Loading and running WebAssembly code.* https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running
[8] 2024. *Node GitHub issue: WebAssembly compiled module cache #36671.* https://github.com/nodejs/node/issues/36671
[9] 2024. *Node.js with WebAssembly.* https://nodejs.org/en/learn/getting-started/nodejs-with-webassembly
[10] 2024. *Optimizing code generated by Emscripten.* https://emscripten.org/docs/optimizing/Optimizing-Code.html
[11] 2024. *Rust: A language empowering everyone to build reliable and efficient software.* https://www.rust-lang.org/what/wasm
[12] 2024. *Twiggy: A code size profiler for Wasm.* https://rustwasm.github.io/twiggy/
[13] 2024. *WABT: The WebAssembly Binary Toolkit.* https://github.com/WebAssembly/wabt
[14] 2024. *Wasmtime – A small and efficient runtime for WebAssembly & WASI.* https://wasmtime.dev/
[15] 2024. *WebAssembly.* https://webassembly.org/

[16] 2024. *WebAssembly JavaScript Interface.* https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface

[17] Ellen Arteca and Alexi Turcotte. 2022. Npm-Filter: Automating the Mining of Dynamic Information from Npm Packages. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22).* Association for Computing Machinery, New York, NY, USA, 304–308. doi:10.1145/3524842.3528501

[18] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. doi:10.1145/3428204

[19] Doehyun Baek, Jakob Getz, Yusung Sim, Daniel Lehmann, Ben Titzer, Sukyoung Ryu, and Michael Pradel. 2024. Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks. In *OOPSLA.*

[20] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly. (2022), 102745. doi:10.1016/j.cose.2022.102745

[21] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A study of inline assembly in solidity smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (Oct. 2022), 27 pages. doi:10.1145/3563328

[22] Bill Budge (Ed.). 2024. *Code caching for WebAssembly developers.* Google. https://v8.dev/blog/wasm-code-caching

[23] ghtopdep 2023-03-28. *ghtopdep.* https://github.com/github-tooling/ghtopdep

[24] Aviral Goel, Pierre Donat-Bouillud, Filip Křikava, Christoph M. Kirsch, and Jan Vitek. 2021. What We Eval in the Shadows: A Large-Scale Study of Eval in R Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 125 (oct 2021), 23 pages. doi:10.1145/3485502

[25] 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. doi:10.1145/3062341.3062363

[26] 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. doi:10.1145/3503921.3503924

[27] 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. doi:10.1145/3442381.3450138

[28] Chika Komiya, Naoto Yanai, Kyosuke Yamashita, and Shingo Okamura. 2023. JABBERWOCK: A Tool for WebAssembly Dataset Generation towards Malicious Website Detection. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W).* 36–39. doi:10.1109/DSN-W58399.2023.00027

[29] 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

[30] 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. doi:10.1145/3297858.3304068

[31] 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 the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23).* Association for Computing Machinery, New York, NY, USA. doi:10.1145/3597926.3598104

[32] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2019) *(DIMVA 2019).* Springer, 23–42. doi:10.1007/978-3-030-22038-9_2

[33] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B. Kent. 2020. Insights into WebAssembly: compilation performance and shared code caching in Node.js. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '20).* IBM Corp., USA, 163–172.

[34] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020).* Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/3426422.3426981

[35] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–78.

[36] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10).* Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/1806596.1806598

[37] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of x86-64 Inline Assembly in C Programs. *SIGPLAN Not.* 53, 3 (March 2018), 84–99. doi:10.1145/3296975.3186418

[38] Quentin Stiévenart and Coen De Roover. 2021. 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.

[39] Andreas Rossberg (Ed.). 2022-03-31. *WebAssembly 1.0 Core Specification.* World Wide Web Consortium (W3C). https://www.w3.org/TR/wasm-core-1/

[40] Andreas Rossberg (Ed.). 2024-04-17. *WebAssembly 2.0 Core Specification.* World Wide Web Consortium (W3C). https://www.w3.org/TR/wasm-core-2/

[41] Yi Yang, Ana Milanova, and Martin Hirzel. 2022. Complex Python Features in the Wild. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR).* 282–293. doi:10.1145/3524842.3528467

[42] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with high risks: a study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19).* USENIX Association, USA, 995–1010.