An Empirical Study of WebAssembly Usage in Node.js

Michelle Thalakottur[1], Max Bernstein[1], Daniel Lehmann[2], Michael Pradel[3], Frank Tip[1]

1

2

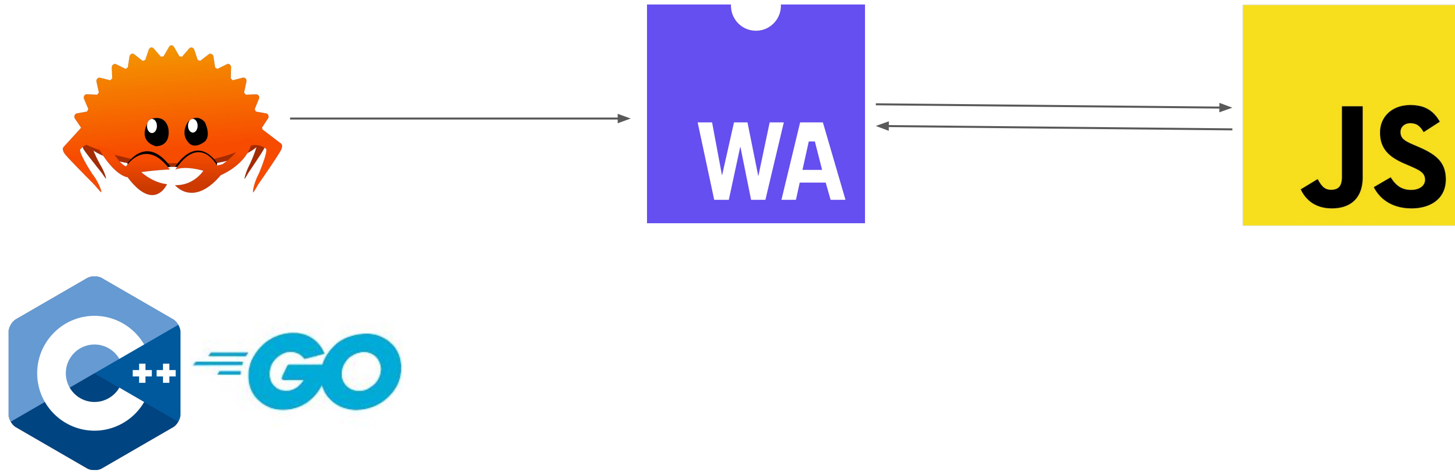
3

WebAssembly

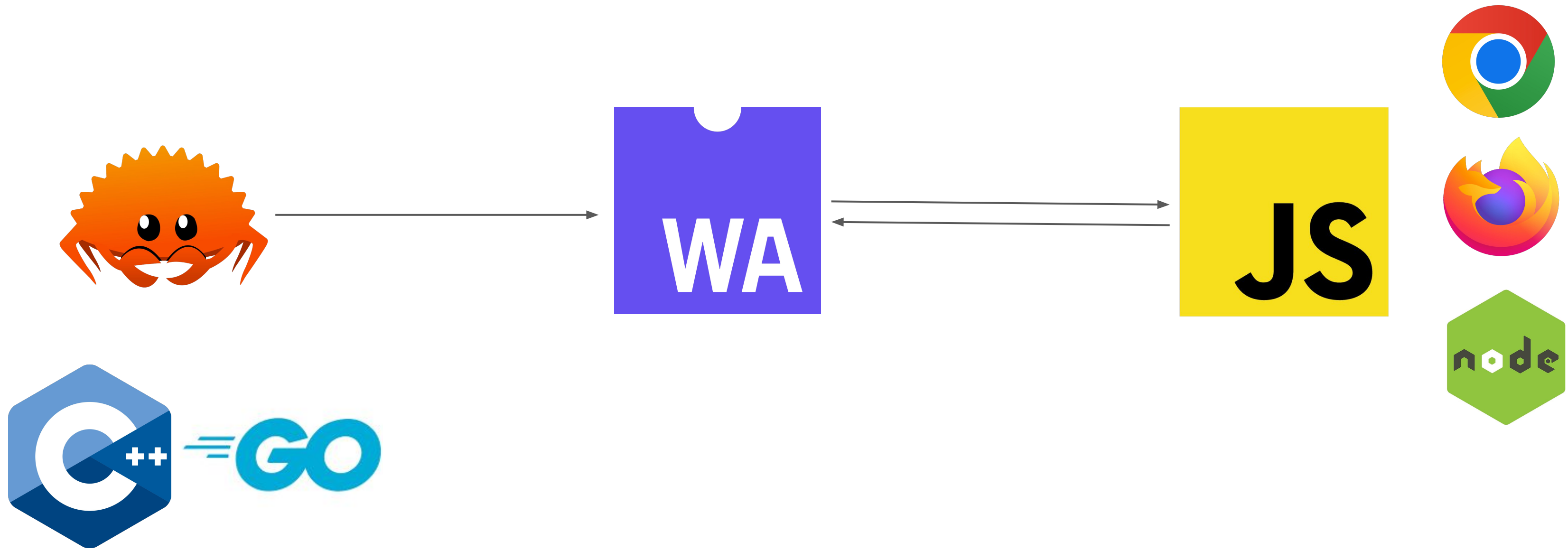
Fast
Compact
Portable

WebAssembly

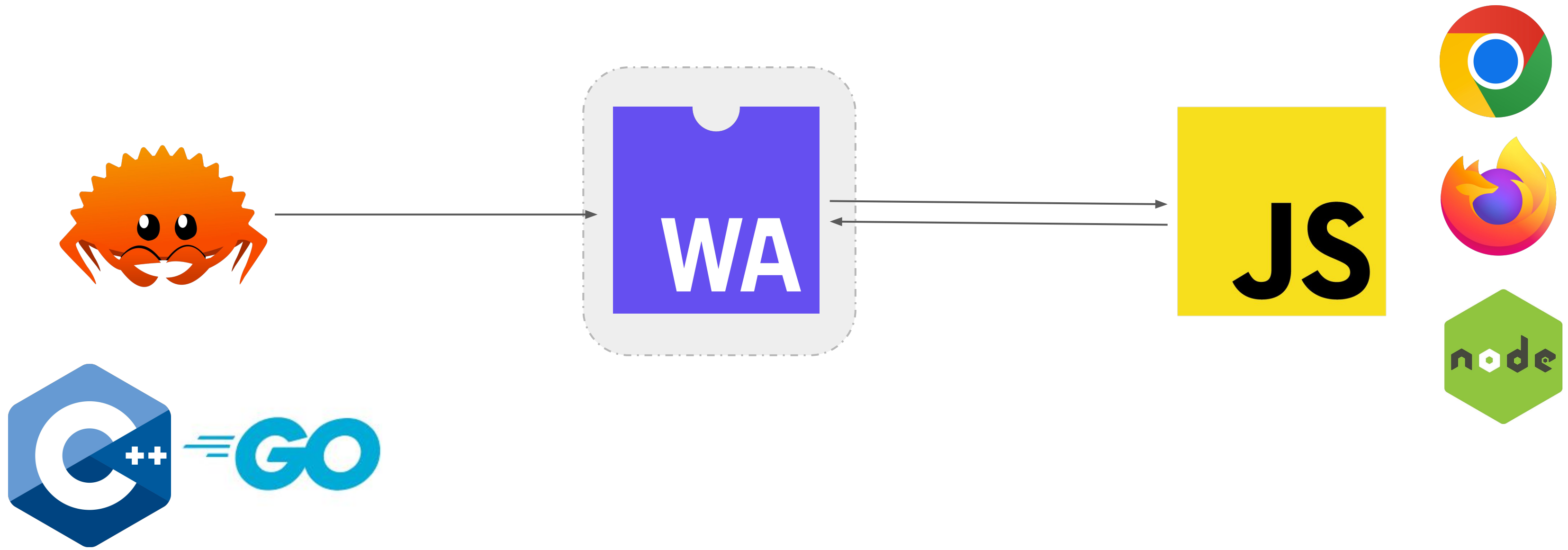
WebAssembly



WebAssembly



WebAssembly



WebAssembly

An Empirical Study of Real-World WebAssembly Binaries

Security, Languages, Use Cases

Aaron Hilbig
aaron@hilbigpost.de
University of Stuttgart
Germany

Daniel Lehmann
mail@dlehmann.eu
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

ABSTRACT

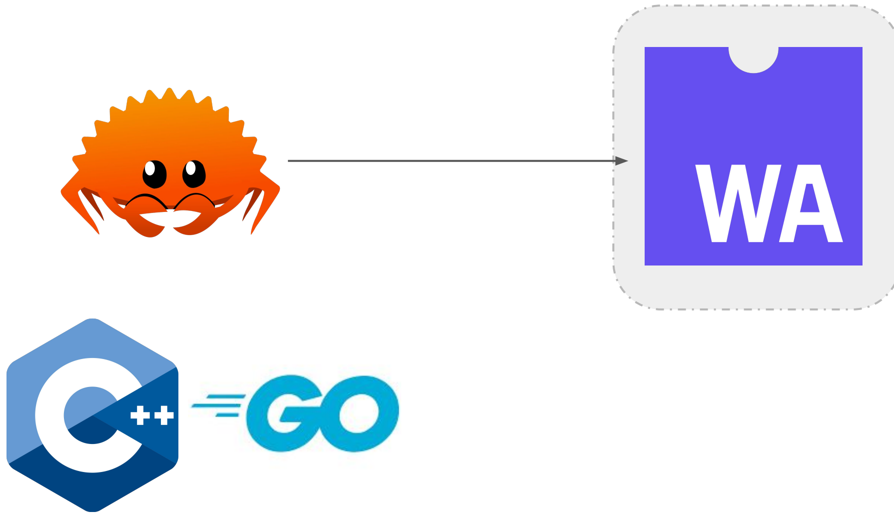
WebAssembly has emerged as a low-level language for the web and beyond. Despite its popularity in different domains, little is known about WebAssembly binaries that occur in the wild. This paper presents a comprehensive empirical study of 8,461 unique WebAssembly binaries gathered from a wide range of sources, including source code repositories, package managers, and live websites. We study the security properties, source languages, and use cases of the binaries and how they influence the security of the WebAssembly ecosystem. Our findings update some previously held assumptions about real-world WebAssembly and highlight problems that call for future research. For example, we show that vulnerabilities that propagate from insecure source languages potentially affect a wide range of binaries (e.g., two thirds of the binaries are compiled from memory unsafe languages, such as C and C++) and that 21% of all binaries import potentially dangerous APIs from their host envi-

supported and available in 93% of all global browser installations as of February 2021.[1] Beyond client-side web applications, WebAssembly is also running on Node.js and even stand-alone runtimes.

Despite its growing popularity, the WebAssembly ecosystem is severely understudied. To date, little is know about how the language is used, for what purposes, and how this affects the security of WebAssembly-based applications. In particular, we are interested in the following research questions:

RQ1: Source languages and tools. WebAssembly is a compilation target, and in principle any programming language can be compiled to it. What languages are actually compiled to WebAssembly, how much do they contribute to the overall population, and what tools are used to produce the binaries? Answering these questions is relevant for understanding the impact of issues that specific source languages may have and for guiding future work toward source languages and toolchains prevalent in practice.

WebAssembly



An Empirical Study of Real-World WebAssembly Binaries

Security, Languages, Use Cases

Modular Abstract Definitional Interpreters for WebAssembly

Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

Quentin Stiévenart
Vrije Universiteit Brussel
Brussels, Belgium
quentin.stievenart@vub.be

David W. Binkley
Loyola University Maryland
Baltimore, MD, USA
binkley@cs.loyola.edu

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

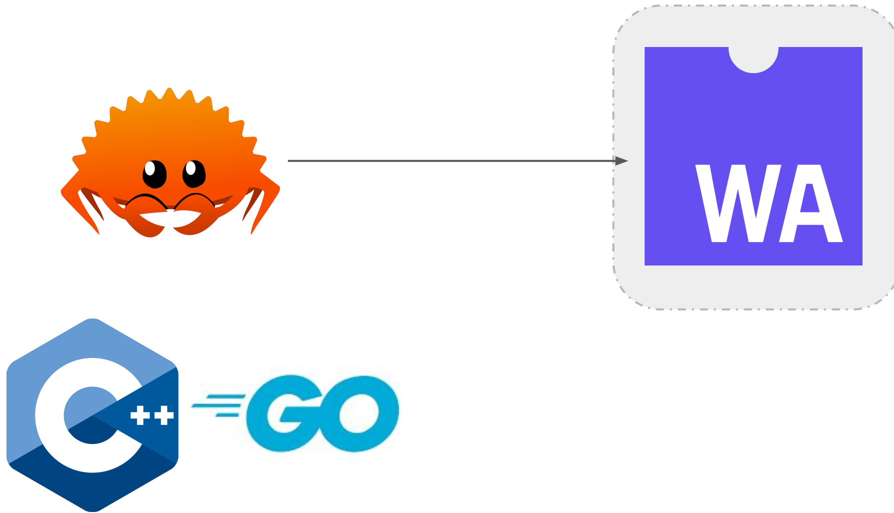
ABSTRACT

The recently introduced WebAssembly standard aims to be a portable compilation target, enabling the cross-platform distribution of programs written in a variety of languages. We propose an approach to *slice* WebAssembly programs in order to enable applications in reverse engineering, code comprehension, and security among others. Given a program and a location in that program, program slicing produces a minimal version of the program that preserves the behavior at the given location. Specifically, our approach is a static, intra-procedural, backward slicing approach that takes into account WebAssembly-specific dependences to identify the instructions of the slice. To do so it must correctly overcome the considerable challenges of performing dependence analysis at the binary level. Furthermore, for the slice to be executable, the approach needs to ensure that the stack behavior of its output complies with WebAssembly's validation requirements. We implemented and eval-

WebAssembly [25] "*is a binary instruction format for a stack-based virtual machine*" [65] designed as a compilation target for high-level languages. The specification of its core has been a W3C standard since December 2019 [49]. WebAssembly was designed for the purpose of embedding binaries in web applications in a portable manner, thereby enabling intensive computations on the web. A 2021 empirical study by Hilbig et al. [30] found use cases on the web as diverse as game engines, natural language processing, and media players. Thanks to its ability to incorporate runtime functions exported by the host environment, WebAssembly has also found usage beyond web applications, broadening the value of analyses for WebAssembly. Examples include desktop applications [63], smart contracts [19], IoT back ends [27], and embedded software [52].

Program slicing [12, 66] is a program decomposition technique that, based on a specific program point called the *slicing criterion*, identifies a subprogram of the code relevant to the slicing

WebAssembly



An Empirical Study of Real-World WebAssembly Binaries

Security, Languages, Use Cases

Modular Abstract Definitional Interpreters for WebAssembly

Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries

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

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

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

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

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

ABSTRACT

WebAssembly is a low-level bytecode format that powers applications and libraries running in browsers, on the server side, and in standalone runtimes. Call graphs are at the core of many interprocedural static analysis and optimization techniques. However, WebAssembly poses some unique challenges for static call graph construction. Currently, these challenges are neither well understood, nor is it clear to what extent existing techniques address them. This paper presents the first systematic study of WebAssembly-specific challenges for static call graph construction and of the state-of-the-art in call graph analysis. We identify and classify 12

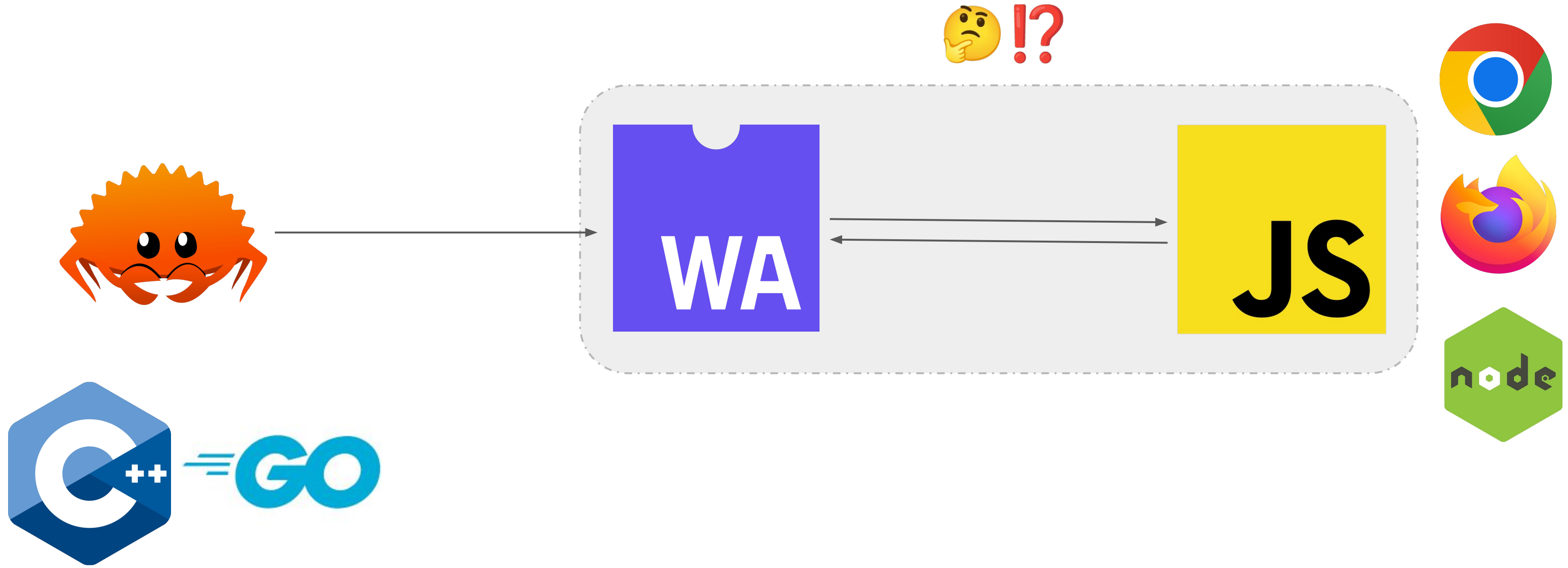
KEYWORDS

WebAssembly, static analysis, call graphs, soundness, precision, debloating

ACM Reference Format:

Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598104

WebAssembly



Contributions

- Novel dataset of 510 executable Node.js packages that exercise 217 unique WebAssembly modules.
- Study Wasm-JS interoperation using static and dynamic analyses.
- We discuss the implications of our findings:
    - We find security risks!
    - We find optimization opportunities!
    - We provide pragmatic assumptions for analyses!

Dataset Collection

NPM package that
uses WebAssembly

Dataset Collection

NPM package that uses WebAssembly

```
-> node_modules
   -> dependency-package
      -> lib.wasm
-> src
   -> foo.js
   -> wasm
      -> wasm-wrapper.js
      -> bar.wasm
```

Dataset Collection

NPM package that uses WebAssembly → (direct usage) → WebAssembly in package source code

```
-> node_modules
   -> dependency-package
      -> lib.wasm
-> src
   -> foo.js
   -> wasm
      -> wasm-wrapper.js
      -> bar.wasm
```

Dataset Collection

NPM package that uses WebAssembly

```
-> node_modules
   -> dependency-package
      -> lib.wasm
-> src
   -> foo.js
   -> wasm
      -> wasm-wrapper.js
      -> bar.wasm
```

direct usage → WebAssembly in package source code

indirect usage → WebAssembly in source code of dependent packages

Dataset Collection

NPM packages that use WebAssembly	*510*
• Packages that directly use WebAssembly	*27*
• Packages that indirectly use WebAssembly	*483*
Statically detected WebAssembly modules	*1,257*
Instantiated WebAssembly modules	*217*

Metrics for the **NoWaSet** dataset

Research Questions

- How do Node.js packages depend on WebAssembly?
- How has WebAssembly usage in NPM packages evolved over time?
- How comprehensively do client packages in the dataset test the WebAssembly modules they depend upon?
- How are JavaScript program analysis and engine developers affected by the presence of WebAssembly?
- What optimization opportunities exist for client packages that use WebAssembly?

Research Questions

- How do Node.js packages depend on WebAssembly?
- **How has WebAssembly usage in NPM packages evolved over time?**
- How comprehensively do client packages in the dataset test the WebAssembly modules they depend upon?
- **How are JavaScript program analysis and engine developers affected by the presence of WebAssembly?**
- **What optimization opportunities exist for client packages that use WebAssembly?**

How has Wasm usage in NPM evolved?

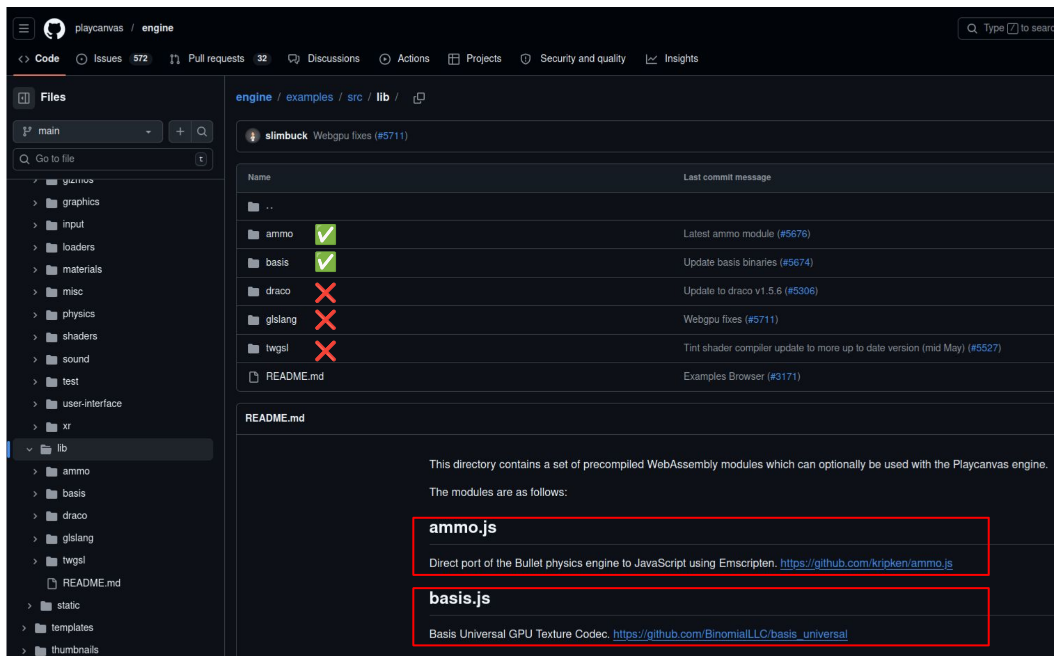
- Manual study over a subset of 50 NPM packages.
- Interested in:
  - When was WebAssembly introduced?
  - Why did packages introduce WebAssembly?
  - What is the source of the WebAssembly binary?
  - How often are WebAssembly binaries updated to keep up with updates in the original source library?

How has Wasm usage in NPM evolved?

- WebAssembly usage has been increasing steadily over time.
- The WebAssembly in packages is more often than not a port of a library in a different language like C/C++/Rust.

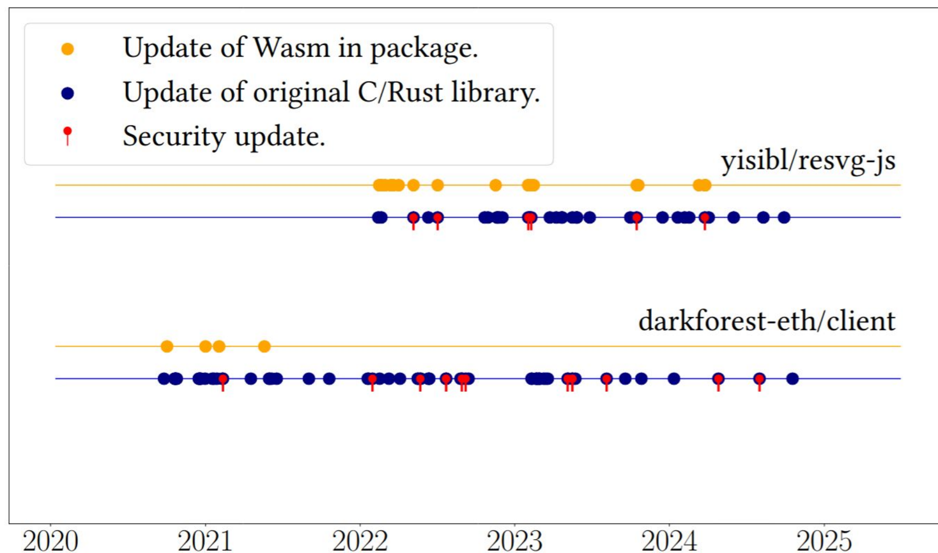
How has Wasm usage in NPM evolved?

- The source of WebAssembly binaries is poorly documented.



How has Wasm usage in NPM evolved?

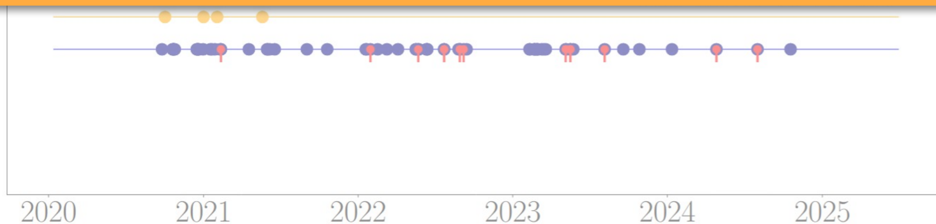
- 66% of packages have *never* updated the WebAssembly binaries to keep up to date with original source library updates.



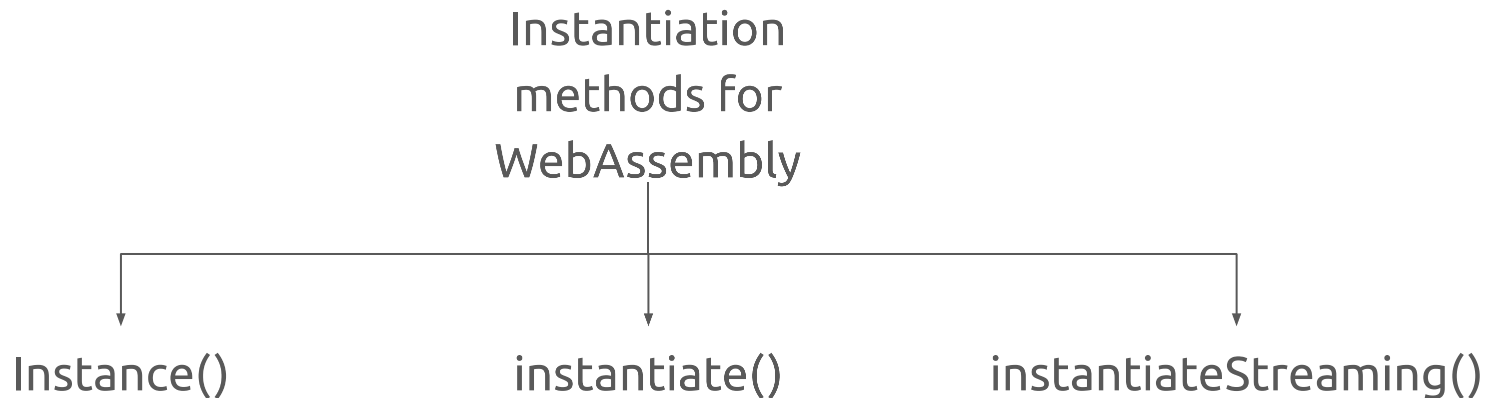
How has Wasm usage in NPM evolved?

- 66% of packages have never updated the WebAssembly binaries to keep up to date with original source library updates.

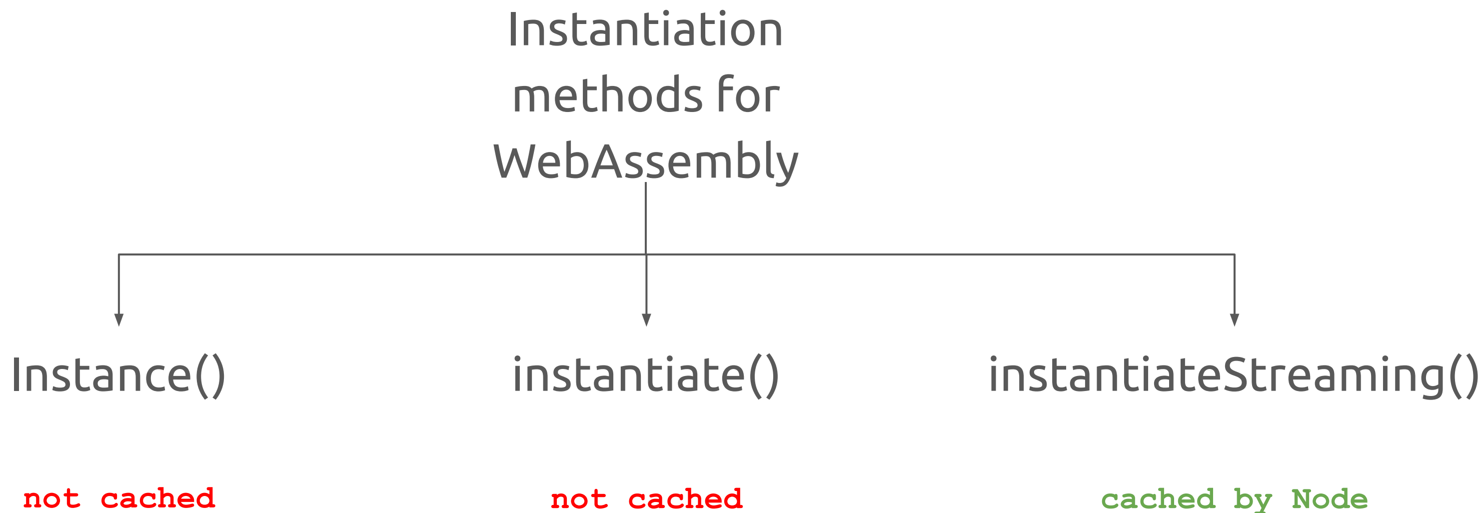
Takeaway: Packages that depend on WebAssembly binaries that are ports of a C/C++/Rust library, are unlikely to update their binaries to keep up with updates and so might miss important security fixes!



Instantiation of Wasm modules



Instantiation of Wasm modules



Instantiation of Wasm modules

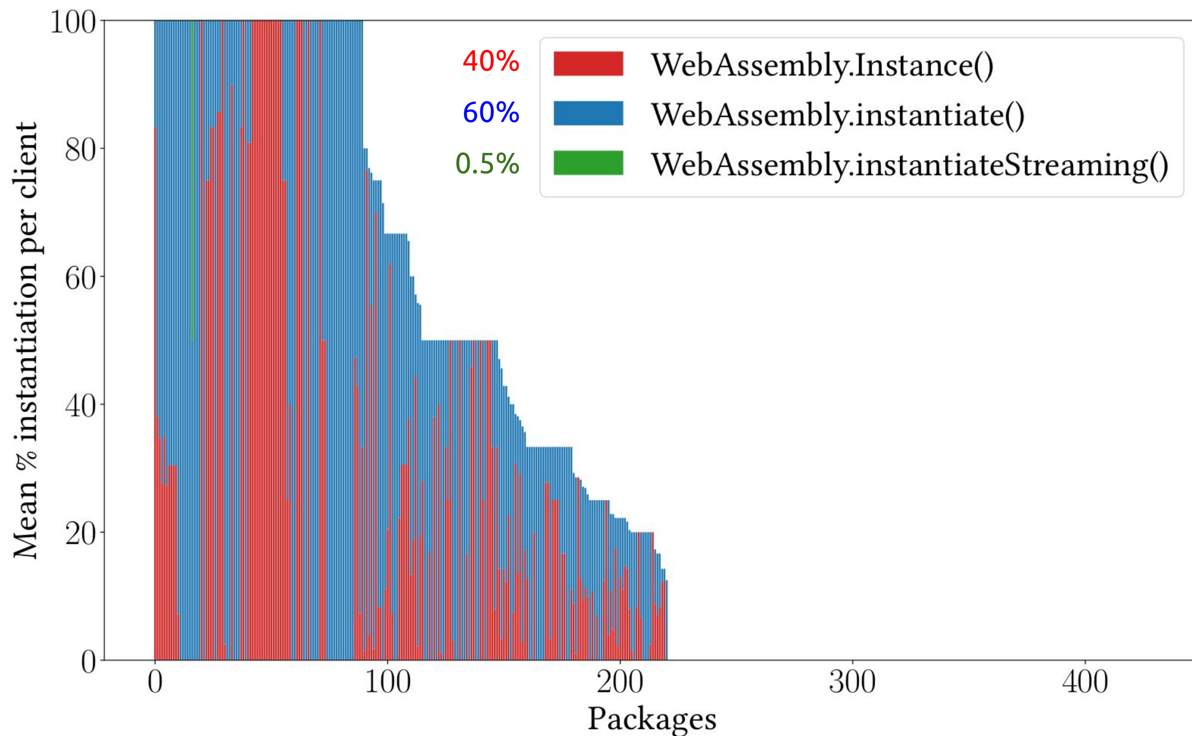
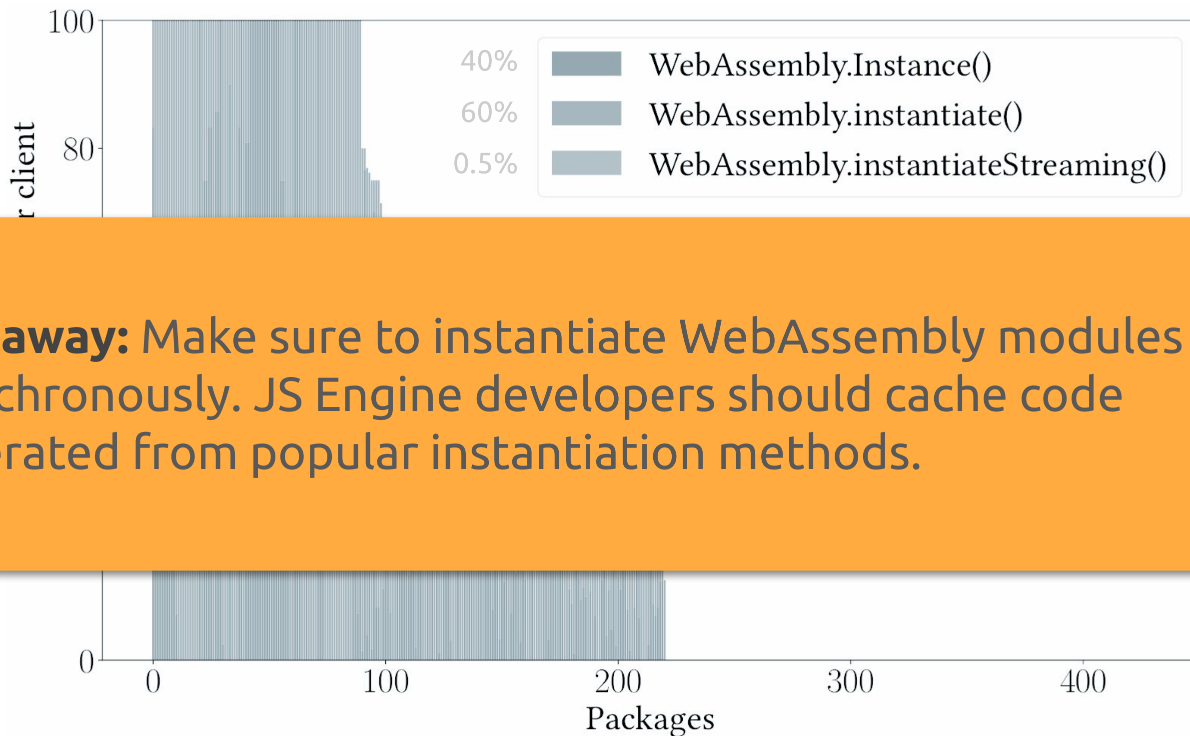


Fig. Instantiation type of WebAssembly modules in packages

Instantiation of Wasm modules

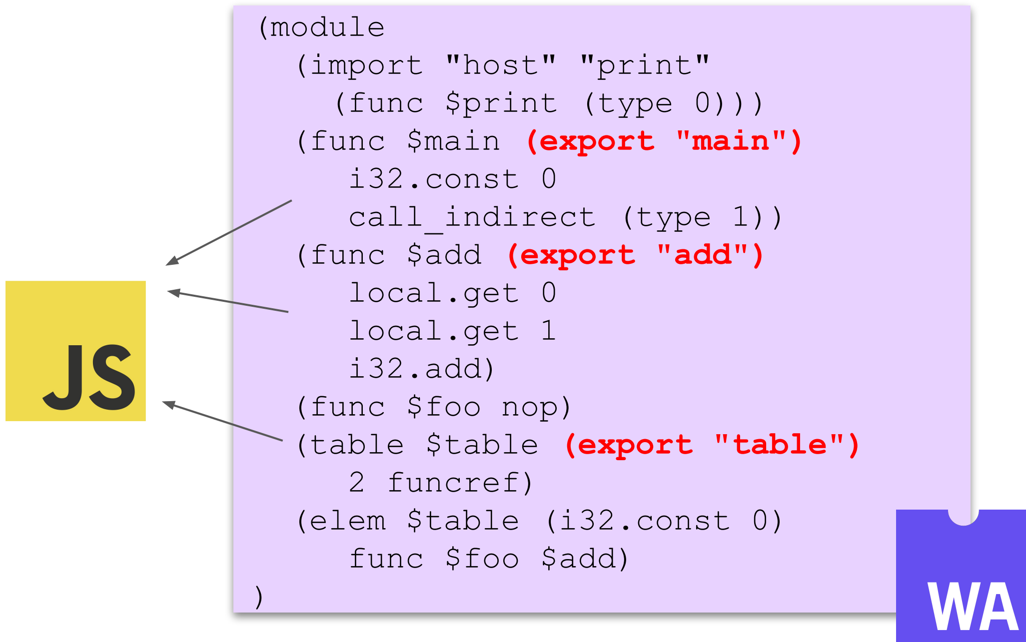


Takeaway: Make sure to instantiate WebAssembly modules asynchronously. JS Engine developers should cache code generated from popular instantiation methods.

Fig. Instantiation type of WebAssembly modules in packages

WebAssembly Exports

WebAssembly exposes functions for a JavaScript client to call via “exports”.



Dynamism in Interoperation

Functions invoked via function pointers (through a function table) rather than the exported functions.

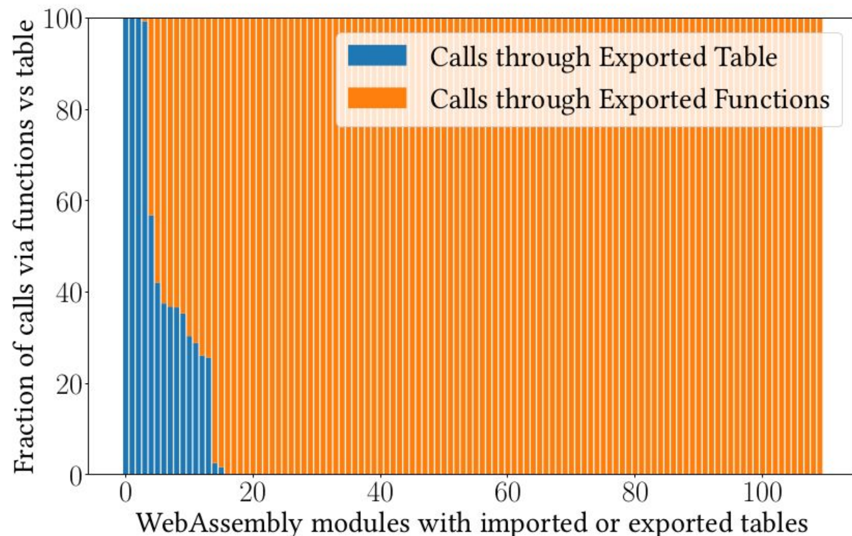
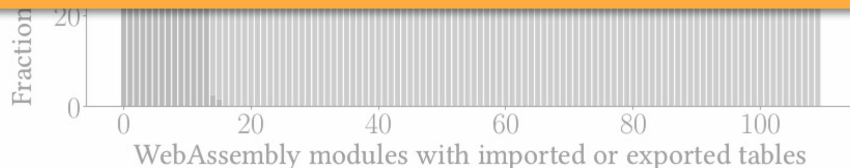


Fig. Fraction of calls from JavaScript through functions vs function tables.

Dynamism in Interoperation

Functions invoked via function pointers (through a function table) rather than the exported functions.



Takeaway: Developers can make the pragmatic assumption that most JavaScript clients are not calling functions through an exported function table.

Fig. Fraction of calls from JavaScript through functions vs function tables.

Optimization Opportunities

Dead Code Elimination

- Remove exported functions in a module that is never called by a client.
- Specialize a library to a specific client use-case.

Optimization Opportunities

Potential of Dead Code Elimination: How many functions are called?

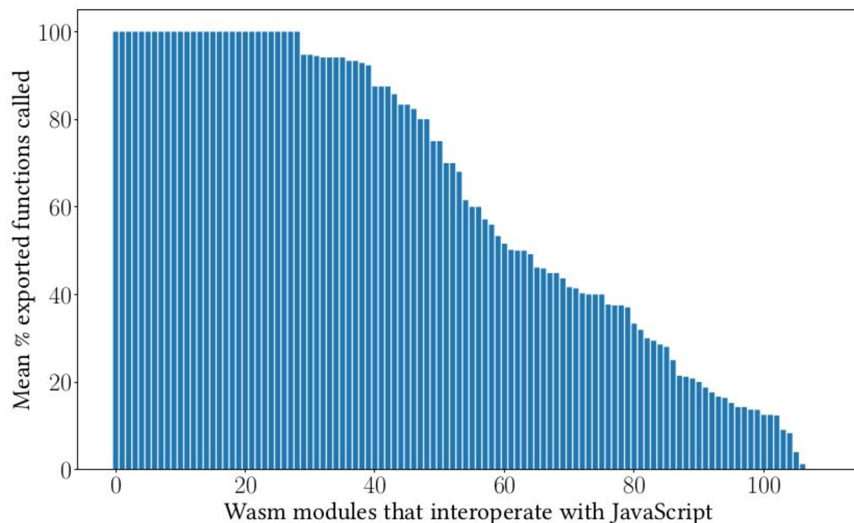


Fig. Mean percentage of functions called by JavaScript

Optimization Opportunities

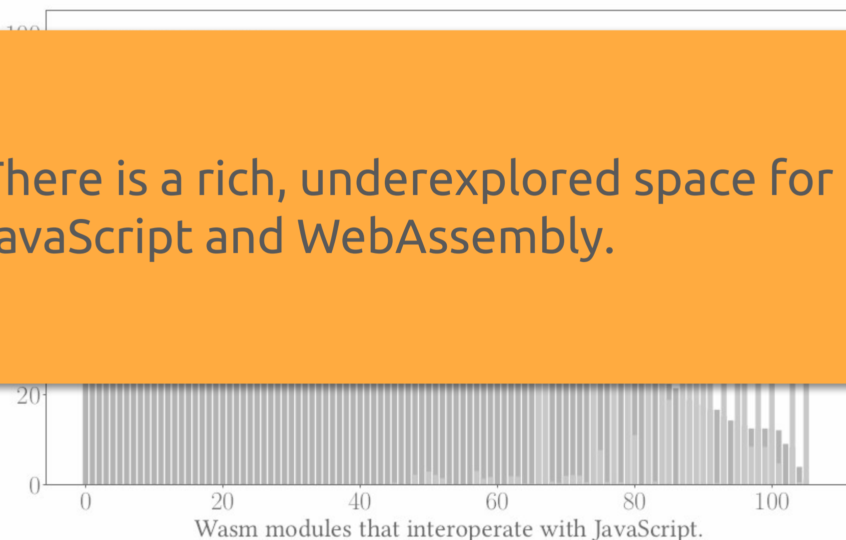
Potential of Dead Code Elimination: using MetaDCE

Mean % exports called
Mean % client-specific reduction wrt baseline size
Wasm modules that interoperate with JavaScript.

Fig. Mean percentage size reduction of binaries

Optimization Opportunities

Potential of Dead Code Elimination: using MetaDCE



Takeaway: There is a rich, underexplored space for program analysis of JavaScript and WebAssembly.

Fig. Mean percentage size reduction of binaries

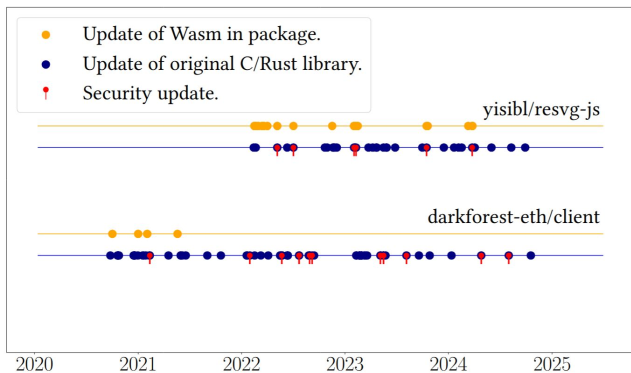
Research Questions

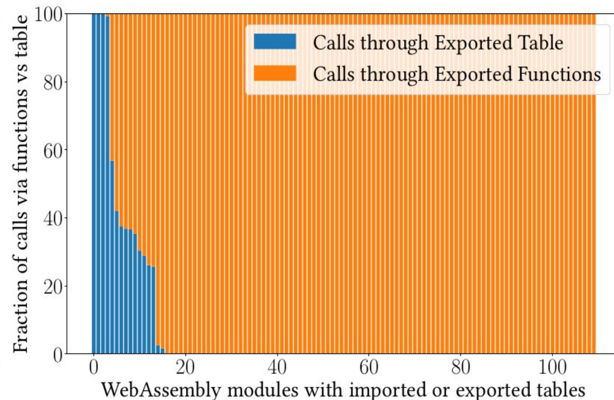
- How do Node.js packages depend on WebAssembly?
- **How has WebAssembly usage in NPM packages evolved over time?**
- How comprehensively do client packages in the dataset test the WebAssembly modules they depend upon?
- **How are JavaScript program analysis and engine developers affected by the presence of WebAssembly?**
- **What optimization opportunities exist for client packages that use WebAssembly?**

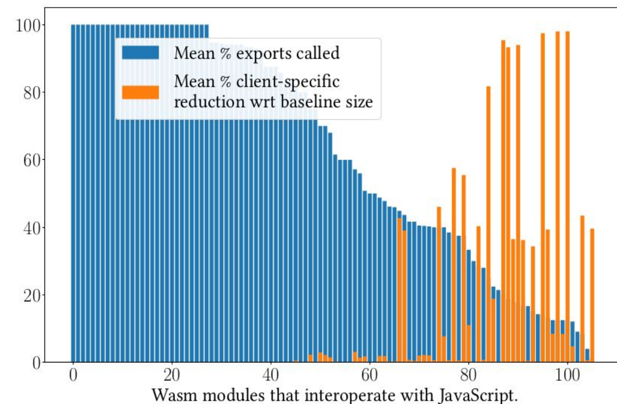
Summary

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







Summary

Metrics for NoWaSet

Metric	Count
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