Empowering Web Applications with WebAssembly: Are We There Yet?

Weihang Wang
University at Buffalo, SUNY
weihangw@buffalo.edu

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678831

***Abstract*—WebAssembly is the newest web standard. It defines a compact bytecode format that allows it to be loaded and executed fast. While WebAssembly is generally believed to be faster than JavaScript, there have been inconsistent results when it comes to showing which code is faster. Unfortunately, insufficient study has been conducted to understand the performance benefits of WebAssembly. In this paper, we investigate how browser engines optimize WebAssembly execution in comparison to JavaScript. In particular, we measure their execution time and memory usage with diverse programs. Our results show that (1) JIT optimization in Chrome significantly impacts JavaScript speed but has no discernible effect on WebAssembly speed; (2) WebAssembly uses much more memory than JavaScript. We hope that our findings can help WebAssembly virtual machine developers uncover optimization opportunities.**

I. Introduction

WebAssembly (abbreviated Wasm) [1] is the newest web standard designed to speed up web applications. It defines a portable and compact bytecode format to serve as a compilation target for other languages such as C, C++, and Rust. Leading companies, such as eBay, Google, and Norton, have recently embraced WebAssembly in a variety of projects (including barcode scanners [2], pattern matching [3], and TensorFlow.js machine learning applications [4]) to improve the speed of services previously developed in JavaScript.

Although WebAssembly is generally believed to be faster than JavaScript, there have been inconsistent findings in practice [5], [6], [7]. For example, eBay developers used WebAssembly to create a barcode scanner, which increased the speed by 50 times over the JavaScript solution [2]. Samsung developers, on the other hand, discovered that when multiplying matrices of specific sizes on the Samsung Internet browser (v7.2.10.12), WebAssembly is slower than JavaScript [8].

Unfortunately, insufficient study has been done to understand WebAssembly's performance benefit over JavaScript. Existing work on WebAssembly performance measurement either focuses on comparing WebAssembly with native code or is limited to one particular type of application. Haas et al. [1] measured the performance of WebAssembly in comparison to asm.js and native code. Jangda et al. [9] analyzed the performance of WebAssembly vs. native code. Sandhu et al. [10] studied the performance of sparse matrix-vector multiplication in WebAssembly, and Herrera et al. [11] measured the performance of numerical programs using WebAssembly.

This paper focuses on comparing WebAssembly with generic JavaScript using diverse benchmarks with multiple inputs. We investigate how browser engines optimize WebAssembly execution comparing to JavaScript. In particular, WebAssembly runtime differs from JavaScript runtime in two aspects. First, WebAssembly programs are delivered as compiled binaries that can be loaded and decoded faster than JavaScript programs, which have to be parsed and compiled at runtime. Second, unlike JavaScript that uses garbage collection, WebAssembly employs a linear memory model that allocates a large chunk of memory at instantiation. In this paper, we measure their performances to answer two research questions:

RQ1. Is WebAssembly always faster than JavaScript?
RQ2. Which is more memory efficient, WebAssembly or JavaScript?

To answer the research questions, we compile 30 widely-used C benchmarks to WebAssembly and JavaScript, and measure their execution time and memory usage on Google Chrome (v79). Our results show that:

1. For small program input, WebAssembly is faster than JavaScript. However, when the input size is increased, more than half of the WebAssembly programs (56.7%) become slower than JavaScript. This is due to the fact that a lengthier execution with repeating loops results in a more aggressive Just-in-time (JIT) compilation on JavaScript. Chrome's JIT optimizations, on the other hand, do not significantly enhance WebAssembly speed.
2. WebAssembly requires substantially more memory than JavaScript in the Chrome browser (v79). This is because of the usage of garbage collection in JavaScript, which dynamically monitors memory allocations to decide when the memory that is no longer in use should be reclaimed. WebAssembly, on the other hand, uses a linear memory model that allocates a big chunk of memory at instantiation and does not reclaim memory automatically.

We hope that our findings and analysis results will help WebAssembly virtual machine developers in improving WebAssembly runtime speed and memory usage.

II. Background

WebAssembly and JavaScript both execute in JavaScript engines. However, the two languages differ significantly in terms of execution model and memory management.

A. Execution Model

JavaScript source code is parsed, optimized, and compiled at runtime. Inside JavaScript engines, JavaScript source code must first be parsed to an abstract syntax tree, which is used to generate bytecode. To speed up JavaScript execution, modern browser engines use Just-in-time (JIT) compilation [12] on the occurrences of frequently executed bytecode to convert it to machine code for direct execution on the hardware.

By contrast, the low-level WebAssembly bytecode does not need to be parsed because it is ready to be compiled into machine code. Moreover, WebAssembly has already gone through the majority of optimizations during compilation. The runtime speed is highly reliant on how well browsers optimize WebAssembly execution.

B. Memory Management

Memory allocation in JavaScript is managed by the garbage collector that automatically monitors memory allocation and identifies when a block of allocated memory is no longer in use and reclaims it. As a result of the automated memory management, JavaScript is relatively memory-efficient. As demonstrated in the experiments (Section IV-B1), the memory required by JavaScript programs remains constant even when dealing with extremely large input.

In contrast to JavaScript's garbage collection, WebAssembly uses a linear memory model [13]. The linear memory is represented as a contiguous buffer of untyped bytes that both WebAssembly and JavaScript can read and modify. When a WebAssembly module is instantiated, a memory instance is created in order to allocate a chunk of linear memory for the module to use and emulate dynamic memory allocations. If the initial memory is used up, the memory instance will be extended to a larger size. When processing a large amount of data, our experiments show that WebAssembly consumes significantly more memory than JavaScript.

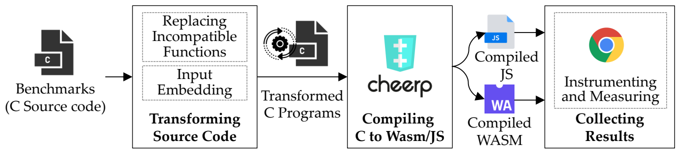


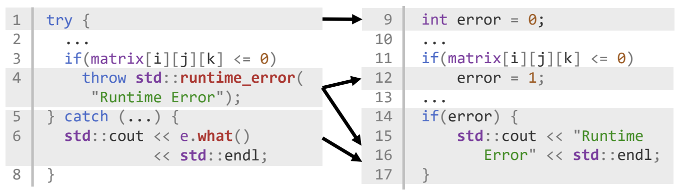
Fig. 1. Overview.

III. Methodology

Fig. 1 shows the procedure we use to measure the performance of WebAssembly and JavaScript. It has three steps: (1) *Transforming Source Code*, (2) *Compiling C to Wasm/JS*, and (3) *Collecting Result*.

A. Transforming Source Code

In the first step, we preprocess the source code by transforming it and replacing incompatible primitives with compliant implementations. This preprocessing step is needed because the Cheerp [14] compiler, which we use to compile C programs to WebAssembly/JavaScript, does not support all C features that mainstream C compilers (such as GCC) do. We handle these features that aren't supported, including `Exception` and `Union`, that prevent us from compiling the C programs to WebAssembly and JavaScript.



```
1  try {
2    ...
3    if(matrix[i][j][k] <= 0)
4      throw std::runtime_error(
         "Runtime Error");
5  } catch (...) {
6    std::cout << e.what()
              << std::endl;
8  }
```

```
 9  int error = 0;
10  ...
11  if(matrix[i][j][k] <= 0)
12      error = 1;
13  ...
14  if(error) {
15      std::cout << "Runtime
16          Error" << std::endl;
17  }
```

Fig. 2. Transforming `try-catch` exception handler.

For example, Cheerp does not support C exceptions properly [15]. In particular, Cheerp eliminates all the `catch` blocks from the `try-catch` statements but leaves the associated `throw` statements in place, resulting in dangling exceptions during runtime. As unhandled exceptions can cause runtime errors, Cheerp provides a workaround solution which forces an abort whenever an exception is thrown through the `-fexceptions` command line option. In practice, however, if any code uses exceptions for purposes other than throwing errors (for example, using exceptions to jump out of loops), Cheerp will not translate the code correctly, and the source code logic after compilation will be wrong. To resolve unsupported exceptions, we transform the source code to a no-exception version. As shown in Fig. 2, we remove the `try-catch` statement and replace a `throw` statement with a variable `error` (at line 9) that stores whether the exception occurs or not. Statements in the `catch` block are copied to the `error` predicate (lines 15-16) which will be executed if the exception occurs.

B. Compiling C to Wasm/JS

After preprocessing, we compile the transformed C programs to WebAssembly and JavaScript using Cheerp. During the compilation, several parameters are used:

- **Input Size.** The value of a program's input that affects the amount of calculations is referred to as its input size (e.g., the dimensions of a matrix multiplication). In our experiment, we compile 30 C benchmark programs chosen from the *PolyBenchC* benchmark suite [16] (Section IV-A). For each benchmark, we use five sets of input: *Extra Small (XS)*, *Small (S)*, *Medium (M)*, *Large (L)*, and *Extra Large (XL)*, as specified in PolyBenchC.
- **Optimization Level.** Cheerp provides several optimization levels. We use optimization level `-O2` in our experiment since it achieves a good balance between execution time, resulting code size, and compilation time [17].
- **Stack/Heap Limit.** By default, Cheerp-compiled WebAssembly programs have a maximum heap size of 8MB and maximum stack size of 1MB. A program that uses heap/stack in excess of this limit will cause runtime errors. To overcome this limit, we use com-

TABLE I
POLYBENCHC BENCHMARK STATISTICS.

Program	cLOC	LOC	Program	cLOC	LOC	Program	cLOC	LOC
covariance	175	958	3mm	229	1,015	trisolv	154	936
correlation	201	984	atax	170	953	deriche	227	1,010
gemm	194	978	bicg	186	969	floyd-warshall	146	928
gemver	215	997	doitgen	176	960	nussinov	495	1,277
gesummv	181	963	mvt	180	962	adi	205	988
symm	194	977	cholesky	170	952	fdtd-2d	214	998
syrk	172	955	durbin	163	945	heat-3d	171	954
syr2k	187	970	gramschmidt	185	974	jacobi-1d	157	940
trmm	171	954	lu	170	952	jacobi-2d	160	943
2mm	214	999	ludcmp	212	994	seidel-2d	150	933

* cLOC excludes benchmark harness.

TABLE II
EXECUTION TIME STATISTICS.

Input Size	SD #^{1}	SD gmean^2	SU #^{3}	SU gmean^4	All gmean^5
Extra-small	0	0x ↓	30	35.30x ↑	35.30x ↑
Small	1	1.53x ↓	29	8.35x ↑	7.67x ↑
Medium	17	1.53x ↓	13	3.68x ↑	1.38x ↑
Large	15	1.67x ↓	15	1.16x ↑	0.83x ↑
Extra-large	17	1.22x ↓	13	1.08x ↑	0.92x ↑

1: # of benchmarks which WebAssembly runs slower than JavaScript. SD is short for the slowdown. 2: Geometric mean for SD. 3: # of benchmarks which WebAssembly runs faster than JavaScript. SU is short for speedup. 4: Geometric mean for SU. 5: Geometric mean for all 30 benchmarks.

piler flags '`cheerp-linear-heap-size`' and '`cheerp-linear-stack-size`' to increase the heap/stack size.

C. Collecting Result

1) Including Wasm/JS within a Webpage: To measure the runtime performance in browsers, we create an HTML webpage that includes the compiled WebAssembly/JavaScript program. This webpage is minimal and includes just the JavaScript program or the JavaScript loader (generated for instantiating WebAssembly) using a '`<script>`' tag to reduce the overhead imposed by other page elements.

2) Measuring Execution Time: We use the JavaScript timer `performance.now()` [18] to measure the execution time, which has a precision of up to microseconds. The timer is included in both the generated JavaScript program and the JavaScript loader, and calls to the timer are placed before and after the target program starts and ends. We run each benchmark ten times and calculate the average.

3) Measuring Memory Usage: We measure memory usage using developer tools, i.e., Chrome DevTools, which contains a heap profiler [19] that shows memory distribution by a page's JavaScript objects and DOM nodes. The memory usage observed includes overhead caused by other browser components such as page renderer. To reduce the overhead imposed by other tasks, we run just one browser tab at a time that executes a single benchmark.

IV. EVALUATION

We measure the performance differences in Google Chrome (v79). The experiments were done on a machine with Intel i7 CPU, 16GB memory, running Ubuntu 18.04.

A. Subject Programs

We compile 30 C benchmarks to WebAssembly/JavaScript and compare their performance differences. Table I lists the 30 C benchmarks. These programs are selected from PolyBenchC (version 4.2.1), which includes compute-intensive programs that we believe may represent some of the use cases for which WebAssembly was designed [20]. For example, it includes programs that perform matrix calculations and graph algorithms which are important kernels for many image processing applications and scientific model simulations.

B. RQ1: Execution Time

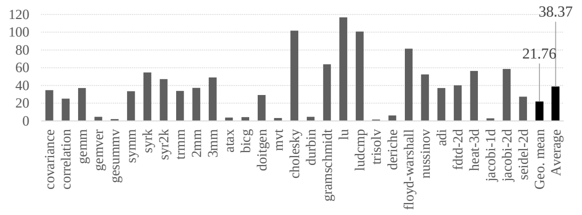
1) Results: The execution time results are shown in Table II. WebAssembly outperforms JavaScript for most of the benchmarks when using XS or S input (100% and 96.7% for XS and S, respectively). Comparing with JavaScript, WebAssembly achieves a 35.30x average speedup for XS inputs and a 7.67x average speedup for S inputs.

However, when the input size is increased to M, there are 17 benchmarks where WebAssembly becomes slower than JavaScript. For example, 'Lu' in WebAssembly was 62.50x and 2.84x faster than JavaScript for XS (N=40) and S (N=120) input, respectively. However, with M input (N=400), it became 2.49x slower. For the other 13 benchmarks, the speed difference between WebAssembly and JavaScript also narrows considerably (3.68x on average). For example, when using XS input, S input, and M input, the WebAssembly version of the '3mm' benchmark is 47.71x, 10.54x, and 1.12x faster than the JavaScript version. When the input is further increased to L or XL, the number of benchmarks where JavaScript outperforms WebAssembly does not increase.

2) JIT Optimization: To investigate why WebAssembly performs worse than JavaScript when inputs are large, we study the impact of Just-In-Time (JIT) compilation. JavaScript engines in modern browsers use JIT compilation to increase the speed of JavaScript by optimizing frequently executed code (e.g., hot-loops) [21]. It is unclear if JIT can substantially improve WebAssembly speed. To understand this correlation, we compare the execution time with JIT enabled and disabled. Specifically, we use the '`--no-opt`' flag [22] and '`--liftoff --no-wasm-tier-up`' flags [23] to disable the JIT optimization (i.e., TurboFan optimizing compiler) for JavaScript and WebAssembly in Chrome.

Fig. 3 shows the performance improvement with JIT, where the y-axis is the ratio of the execution time without JIT to the execution time with JIT. The last two bars are the geometric mean and average. As can be seen, JIT-enabled JavaScript outperforms JIT-disabled JavaScript 38.37 times on average. By contrast, most of the WebAssembly programs have performance improvement ratios close to one, suggesting that there is no substantial difference in performance with and without JIT.

Takeaway 1: JIT optimization in Chrome significantly impacts JavaScript speed but has no discernible effect on WebAssembly speed.


(a) JavaScript – PolyBenchC

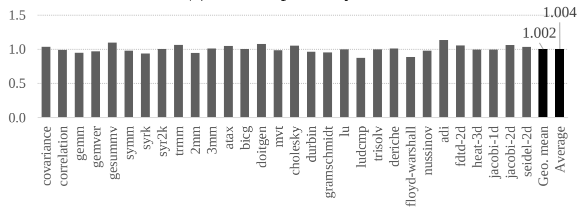

(b) WebAssembly – PolyBenchC

Fig. 3. Performance improvement by JIT.

TABLE III
AVERAGE MEMORY USAGE (IN KB).

Input Size	JavaScript	WebAssembly
Extra-small	885.5	2,020.1
Small	883.5	2,123.7
Medium	883.7	3,363.1
Large	884.3	36,168.3
Extra-large	885.1	143,899.5

C. RQ2: Memory Usage

The memory usage statistics are shown in Table III. As can be seen, the memory usage of JavaScript does not change substantially regardless of the input (between 883.5KB and 885.5KB). By contrast, WebAssembly programs consume substantially more memory for larger inputs. On average, WebAssembly programs use ≈36MB of memory with L inputs and ≈144MB of memory with XL inputs. This is due to the fact that WebAssembly does not support garbage collection [24]. When a WebAssembly module was instantiated, a large chunk of linear memory was initialized to simulate memory allocations. If the initial memory is used up, rather than reclaiming memory that is no longer in use, the linear memory is expanded to a larger size. JavaScript, on the other hand, uses garbage collection, which dynamically monitors memory allocations and reclaims unneeded memory.

Takeaway 2: JavaScript is more memory-efficient than WebAssembly due to garbage collection, which WebAssembly presently does not support.

V. RELATED WORK

Our work is closely related to WebAssembly performance measurement. Haas et al. [1] measured the performance of WebAssembly in comparison to asm.js and native code. Jangda et al. [9] analyzed the performance of WebAssembly vs. native code. Sandhu et al. [10] studied the performance of sparse matrix-vector multiplication in WebAssembly, and Herrera et al. [11] measured the performance of numerical programs using WebAssembly. Existing work either focuses on comparing WebAssembly with native code or is limited to one particular type of application. By contrast, our work compares WebAssembly and generic JavaScript using diverse benchmarks with multiple inputs.

There have been prior works on WebAssembly analysis tools, protections, and studies [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39]. Wasabi [26] is the first general-purpose framework for dynamically analyzing WebAssembly. Lehmann et al. [27] analyzed how vulnerabilities in memory-unsafe source languages are exploitable in WebAssembly binaries. Swivel [28] is a new compiler framework for hardening WebAssembly against Spectre attacks. Musch et al. [25] studied the prevalence of WebAssembly, and Hilbig et al. [33] analyzed security properties, source languages, and use cases of real-world WebAssembly binaries.

VI. LIMITATION AND FUTURE WORK

Our study is potentially subject to several threats, including the representativeness of the benchmarks used and the generalizability of the results. WebAssembly was designed to be utilized in a range of applications such as cryptographic libraries, games, image processing, arithmetic computation, and others [25], [20]. While we believe the benchmarks we used may represent some common WebAssembly use cases, we do not measure large standalone applications such as games in the comparison. Cheerp is unable to build these applications due to the complexity of its source code. We plan to address the incompatibility issues in the future by changing the compiler or rewriting the source code. Another threat concerns the generalization of the performance results. The benchmarks used in the study were evaluated on Google Chrome, one of the most popular web browsers. As a result, the findings of this study do not represent the performance of other popular browsers such as Mozilla Firefox and Microsoft Edge. We plan to compare the performance difference across various browsers in the future.

Our future work includes four directions. First, WebAssembly runtime environments play a critical role in performance. Thus, we plan to study the performance difference on various browsers and platforms. Second, we find that compilers generating WebAssembly programs can impact the runtime performance, especially the optimization techniques applied. Next, we want to investigate the impact of compilers used in generating WebAssembly binaries. Third, our current dataset is limited to simple C benchmarks with a few hundred LOC. We will add complex real-world applications in the experiments. Fourth, we plan to investigate how to optimize WebAssembly runtime to improve its speed and memory efficiency.

VII. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was partially supported by NSF 2047980 and a Mozilla Research Grant (2019). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] 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: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363

[2] S. Padmanabhan and P. Jha, "WebAssembly at eBay: A Real-World Use Case," 2020. [Online]. Available: https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/

[3] R. Hill, "uBlock Origin," 2019. [Online]. Available: https://github.com/gorhill/uBlock

[4] D. Smilkov, N. Thorat, and A. Yuan, "Introducing the WebAssembly backend for TensorFlow.js," 2020. [Online]. Available: https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html

[5] Vladimir, "WebAssembly vs. the world. Should you use WebAssembly?" 2018. [Online]. Available: https://blog.sqreen.com/webassembly-performance/

[6] Stack Overflow Contributor Blindman67, "Why is webAssembly function almost 300 time slower than same JS function," 2018. [Online]. Available: https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function

[7] Stack Overflow Contributor ColinE, "Why is my WebAssembly function slower than the JavaScript equivalent?" 2017. [Online]. Available: https://stackoverflow.com/questions/46331830/why-is-my-webassembly-function-slower-than-the-javascript-equivalent/46500236#46500236

[8] W. Chen, "Performance Testing WebAssembly vs JavaScript," 2018. [Online]. Available: https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875

[9] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so Fast: Analyzing the Performance of Webassembly vs. Native Code," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 107–120.

[10] P. Sandhu, D. Herrera, and L. Hendren, "Sparse Matrices on the Web: Characterizing the Performance and Optimal Format Selection of Sparse Matrix-Vector Multiplication in JavaScript and WebAssembly," in *Proceedings of the 15th International Conference on Managed Languages amp; Runtimes*, ser. ManLang '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3237009.3237020

[11] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices," *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2*, 2018.

[12] The Chromium Project, "V8 JavaScript Engine," 2020. [Online]. Available: https://v8.dev/

[13] Mozilla, "WebAssembly Memory," 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/WebAssembly/Memory

[14] L. Technologies, "Cheerp — c/c++ to webassembly compiler," 2020. [Online]. Available: https://leaningtech.com/pages/cheerp.html

[15] C. Contributors., "Cheerp FAQs," 2020. [Online]. Available: https://github.com/leaningtech/cheerp-meta/wiki/FAQs-(Frequently-asked-questions)

[16] L.-N. Pouchet, U. Bondugula, and T. Yuki, "PolyBench/C 4.2. Polyhedral Benchmark Suite," 2016.

[17] The Clang Team, "clang - the Clang C, C++, and Objective-C compiler — Clang 11 documentation," 2020. [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html#cmdoption-o0

[18] MDN., "MDN Web Docs - performance.now()." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[19] C. Developers., "Record heap snapshots." [Online]. Available: https://developer.chrome.com/docs/devtools/memory-problems/heap-snapshots/

[20] WebAssembly Contributors, "Webassembly Use Cases," 2020. [Online]. Available: https://webassembly.org/docs/use-cases/

[21] Google, "TurboFan," 2021. [Online]. Available: https://v8.dev/docs/turbofan

[23] V8, "Webassembly compilation pipeline," 2021. [Online]. Available: https://v8.dev/docs/wasm-compilation-pipeline

[22] J. Gruber, "JIT-less V8," 2021. [Online]. Available: https://v8.dev/blog/jitless

[24] WebAssembly Group, "WebAssembly/design," 2020. [Online]. Available: https://github.com/WebAssembly/design/blob/master/FutureFeatures.md

[25] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "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*. Springer, 2019, pp. 23–42.

[26] 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, 2019, p. 1045–1058. [Online]. Available: https://doi.org/10.1145/3297858.3304068

[27] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[28] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1433–1450. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[29] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290390

[30] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position Paper: Progressive Memory Safety for WebAssembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337167.3337171

[31] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a Practical Path to Library Sandboxing," *CoRR*, vol. abs/1912.02285, 2019. [Online]. Available: http://arxiv.org/abs/1912.02285

[32] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust, but verify: SFI safety for native-compiled Wasm," in *NDSS*. Internet Society, 2021.

[33] 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, 2021, p. 2696–2708. [Online]. Available: https://doi.org/10.1145/3442381.3450138

[34] A. Romano, Y. Zheng, and W. Wang, "MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1129–1140. [Online]. Available: https://doi.org/10.1145/3324884.3416580

[35] A. Romano and W. Wang, "WasmView: Visual Testing for Webassembly Applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 13–16. [Online]. Available: https://doi.org/10.1145/3377812.3382155

[36] A. Romano and W. Wang, "WASim: Understanding WebAssembly Applications through Classification," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1321–1325. [Online]. Available: https://doi.org/10.1145/3324884.3415293

[37] A. Szanto, T. Tamm, and A. Pagnoni, "Taint Tracking for WebAssembly," *CoRR*, vol. abs/1807.08349, 2018. [Online]. Available: http://arxiv.org/abs/1807.08349

[38] W. Fu, R. Lin, and D. Inge, "TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly," *CoRR*, vol. abs/1802.01050, 2018. [Online]. Available: http://arxiv.org/abs/1802.01050

[39] H. Jeong, J. Jeong, S. Park, and K. Kim, "WATT : A novel web-based toolkit to generate WebAssembly-based libraries and applications," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2018, pp. 1–2.