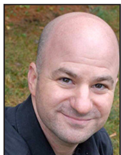
Not So Fast

Analyzing the Performance of WebAssembly vs. Native Code

ABHINAV JANGDA, BOBBY POWERS, EMERY BERGER, AND ARJUN GUHA

Abhinav Jangda is a PhD student in the College of Information and Computer Sciences at the University of Massachusetts Amherst. For his research, Abhinav focuses on designing programming languages and compilers. He loves to write and optimize high performance code in his leisure time.
aabhinav@cs.umass.edu

Bobby Powers is a PhD candidate at the College of Information and Computer Sciences at the University of Massachusetts Amherst (in the PLASMA lab), and he is a Software Engineer at Stripe. His research spans systems and programming languages, with a focus on making existing software more efficient, more secure, and usable in new contexts.
bobbypowers@gmail.com



Emery Berger is a Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts), and he is a regular visiting researcher at Microsoft Research, where he is currently on sabbatical. His research interests span programming languages and systems, with a focus on systems that transparently increase performance, security, and reliability.
emery@cs.umass.edu

WebAssembly is a new low-level programming language, supported by all major browsers, that complements JavaScript and is designed to provide performance parity with native code. We developed Browsix-Wasm, a "UNIX kernel in a web page" that works on unmodified browsers and supports programs compiled to WebAssembly. Using Browsix-Wasm, we ran the SPEC CPU benchmarks in the browser and investigated the performance of WebAssembly in detail.

Web browsers have become the most popular platform for running user-facing applications, and, until recently, JavaScript was the only programming language supported by all major web browsers. Beyond its many quirks and pitfalls from the perspective of programming language design, JavaScript is also notoriously difficult to execute efficiently. Programs written in JavaScript typically run significantly slower than their native counterparts.

There have been several attempts at running native code in the browser instead of JavaScript. ActiveX was the earliest technology to do so, but it was only supported in Internet Explorer and required users to trust that ActiveX plugins were not malicious. Native Client [2] and Portable Native Client [3] introduced a sandbox for native code and LLVM bitcode, respectively, but were only supported in Chrome.

Recently, a group of browser vendors jointly developed the WebAssembly (Wasm) standard [4]. WebAssembly is a low-level, statically typed language that does not require garbage collection and supports interoperability with JavaScript. WebAssembly's goal is to serve as a portable compiler target that can run in a browser. To this end, WebAssembly is designed not only to sandbox untrusted code, but to be fast to compile, fast to run, and portable across browsers and architectures.

WebAssembly is now supported by all major browsers and has been swiftly adopted as a back end for several programming languages, including C, C++, Rust, Go, and several others. A major goal of WebAssembly is to be faster than JavaScript. For example, initial results showed that when C programs are compiled to WebAssembly instead of JavaScript, they run 34% faster in Chrome [4]. Moreover, on a suite of 24 C program benchmarks that were compiled to WebAssembly, seven were less than 10% slower than native code, and almost all were less than twice as slow as native code. We recently re-ran these benchmarks and found that WebAssembly's performance had improved further: now 13 out of 24 benchmarks are less than 10% slower than native code.

These results appear promising, but they beg the question: are these 24 benchmarks really representative of WebAssembly's intended use cases?

The Challenge of Benchmarking WebAssembly

The 24 aforementioned benchmarks are from the PolybenchC benchmark suite [5], which is designed to measure the effect of polyhedral loop optimizations in compilers. Accordingly, they constitute a suite of small scientific computing kernels rather than full-fledged

Arjun Guha is an Assistant Professor in the College of Information and Computer Sciences at the University of Massachusetts Amherst, where he co-directs the PLASMA lab (Programming Languages and Systems at Massachusetts). His research interests include web programming, web security, network configuration languages, and system configuration languages.
arjunguha@umass.edu

applications. In fact, each benchmark is roughly 100 lines of C code. WebAssembly is meant to accelerate scientific kernels, but it is explicitly designed for a wider variety of applications. The WebAssembly documentation highlights several intended use cases, including scientific kernels, image editing, video editing, image recognition, scientific visualization, simulations, programming language interpreters, virtual machines, and POSIX applications. In other words, WebAssembly's solid performance on scientific kernels does not imply that it will also perform well on other kinds of applications.

We believe that a more comprehensive evaluation of WebAssembly should use established benchmarks with a diverse collection of large programs. The SPEC CPU benchmarks meet this criterion, and several of the SPEC benchmarks fall under WebAssembly's intended use cases. For example, there are eight scientific applications, two image and video processing applications, and all the benchmarks are POSIX applications.

Unfortunately, it is not always straightforward to compile a native program to WebAssembly. Native programs, including the SPEC CPU benchmarks, require operating system services, such as a file system, synchronous I/O, processes, and so on, which WebAssembly does not itself provide.

Despite its name, WebAssembly is explicitly designed to run in a wide variety of environments, not just the web browser. To this end, the WebAssembly specification imposes very few requirements on the execution environment. A WebAssembly module can import externally defined functions, including functions that are written in other languages (e.g., JavaScript). However, the WebAssembly specification neither prescribes how such imports work, nor prescribes a standard library that should be available to all WebAssembly programs.

There is a separate standard [7] that defines a JavaScript API to WebAssembly that is supported by all major browsers. This API lets JavaScript load and run a Wasm module, and allows JavaScript and Wasm functions to call each other. In fact, the only way to run WebAssembly in the browser is via this API, so all WebAssembly programs require at least a modicum of JavaScript to start. Using this API, a WebAssembly program can rely on JavaScript for I/O operations, including drawing to the DOM, making networking requests, and so on. However, this API also does not prescribe a standard library.

Emscripten [6] is the de facto standard toolchain for compiling C/C++ applications to WebAssembly. The Emscripten runtime system, which is a combination of JavaScript and WebAssembly, implements a handful of straightforward system calls, but it does not scale up to larger applications. For example, the default Emscripten file system (MEMFS) loads the entire file-system image in memory before execution. For the SPEC benchmarks, the file system is too large to fit into memory. The SPEC benchmarking harness itself requires a file system, a shell, the ability to spawn processes, and other UNIX facilities, none of which Emscripten provides.

Most programmers overcome these limitations by modifying their code to avoid or mimic missing operating system services. Modifying well-known benchmarks, such as SPEC CPU, would not only be time-consuming but would also pose a serious threat to the validity of any obtained results.

Our Contributions

To address these challenges, we developed Browsix-Wasm, which is a simulated UNIX-compatible kernel for the browser. Browsix-Wasm is written in JavaScript (compiled from TypeScript) and provides a range of operating system services to Wasm programs, including processes, files, pipes, and blocking I/O. We have engineered Browsix-Wasm to be fast, which is necessary both for usability and for benchmarking results to be valid [1].

Using Browsix-Wasm, we conducted the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that Wasm is faster than JavaScript (1.3× faster on average). However, contrary to prior work, we found a substantial gap between WebAssembly and native performance. Code compiled to Wasm ran on average 1.55× slower in Chrome and 1.45× slower in Firefox.

Digging deeper, we conducted a forensic analysis of these results with the aid of CPU performance counters to identify the root causes of this performance gap. For example, we found that Wasm produced code with more loads and stores, more branches, and more L1 cache misses than native code. It is clear that some of the issues that we identified can be addressed with engineering effort. However, we also identified more fundamental performance problems that appeared to arise from the design of WebAssembly, which will be harder to address. We provided guidance to help WebAssembly implementers focus their optimization efforts in order to close the performance gap between WebAssembly and native code.

In the rest of this article, we present the design and implementation of Browsix-Wasm and give an overview of our experimental results. This article is based on a conference paper that appeared at the 2019 USENIX Annual Technical Conference, which presents Browsix-Wasm, our experiments, our analysis, and related work in detail [1].

Overview of Browsix-Wasm

Browsix-Wasm mimics a UNIX kernel within a web page with no changes or extensions needed to a browser. Browsix-Wasm supports multiple processes, pipes, and the file system. At a high-level, the majority of the kernel, which is written in JavaScript, runs on the main thread of the page, whereas each WebAssembly process runs within a WebWorker, which runs concurrently with the main thread. In addition, each WebWorker also runs a small amount of JavaScript that is necessary to start the WebAssembly process and to manage process-to-kernel communication for system calls.

In an ordinary operating system, the kernel has direct access to each process's memory, which makes it straightforward to transfer data to and from a process (e.g., to read and write files). Web browsers allow a web page to share a block of memory between the main thread and WebWorkers using the SharedArrayBuffer API. In principle, a natural way to build Browsix-Wasm would be to have each WebAssembly process share its memory with the kernel as a SharedArrayBuffer.

Unfortunately, there are several issues with this approach. First, a SharedArrayBuffer cannot be grown, which precludes programs from growing the heap on demand. Second, browsers impose hard memory limits on each JavaScript thread (2.2 GB in Chrome), and thus the total memory available to Browsix-Wasm would be 2.2 GB across all processes. Finally, the most fundamental problem is that WebAssembly programs cannot access SharedArrayBuffer objects.

Instead, Browsix-Wasm adopts a different approach. Within each WebWorker, Browsix-Wasm creates a small (64 MB) SharedArrayBuffer that it shares with the kernel. When a system call references strings or buffers in the process's heap (e.g., writev or stat), the runtime system copies data from the process memory to the shared buffer and sends a message to the kernel with locations of the copied data in auxiliary memory. Similarly, when a system call writes data to the auxiliary buffer (e.g., read), its runtime system copies the data from the shared buffer to the process memory at the memory specified. Moreover, if a system call specifies a buffer in process memory for the kernel to write to (e.g., read), the runtime allocates a corresponding buffer in auxiliary memory and passes it to the kernel. If a system call must transfer more than 64 MB, Browsix-Wasm breaks it up into several operations that only transfer 64 MB of data. The cost of these memory copy operations is dwarfed by the overall cost of the system call invocation, which involves sending a message between process and kernel JavaScript contexts.

Using Browsix-Wasm, we are able to run the SPEC benchmarks and the SPEC benchmarking harness unmodified within the browser. The only portions of our toolchain that work outside the browser are (1) capturing performance counter data, which cannot be done within a browser, and (2) validating benchmark results, which we do outside the browser to avoid errors.

Performance Evaluation

Browsix-Wasm provided what we needed to compile the SPEC benchmarks to WebAssembly, run them in the browser, and collect performance counter data. We ran all benchmarks on a 6-Core Intel Xeon E5-1650 v3 CPU with hyperthreading and 64 GB of RAM. We used Google Chrome 74.0 and Mozilla Firefox 66.0. Our ATC paper describes the experimental setup and evaluation methodology in more detail.

Reproducing Results with PolybenchC

Although our goal was to conduct a performance evaluation with the SPEC benchmarks, we also sought to reproduce the results by Haas et al. [4] that used PolybenchC. We were able to run these benchmarks (which do not make system calls): the most recent implementations of WebAssembly are now faster than they were two years ago.

Measuring the Cost of Browsix-Wasm

It is important to rule out the possibility that any slowdown that we report is due to poor performance by the Browsix-Wasm

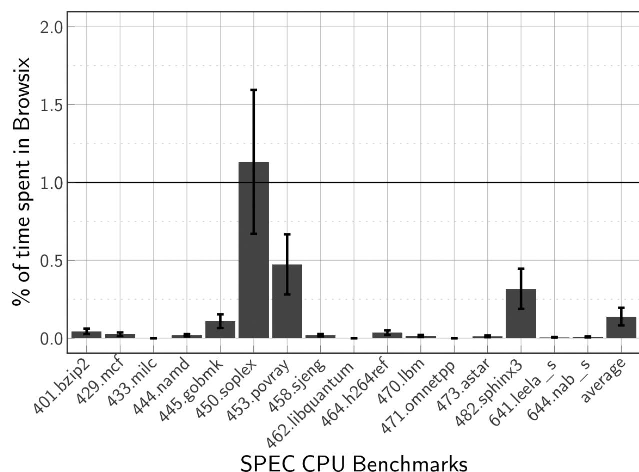


Figure 1: Percentage of time spent (in %) in Browsix-Wasm calls in Firefox

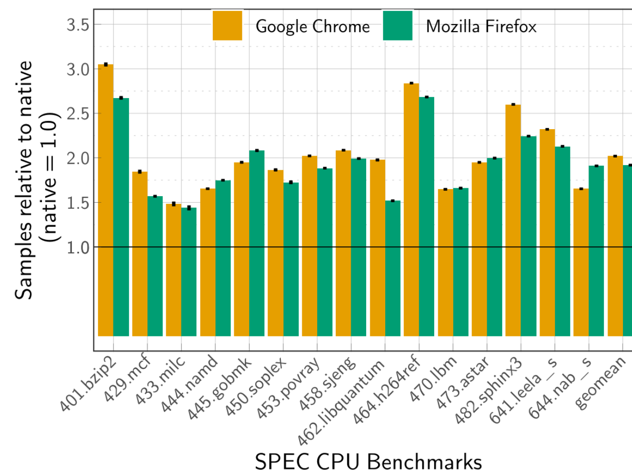


Figure 2: Ratio of the number of load instructions retired by WebAssembly over native code

kernel. In particular, since Browsix-Wasm implements system calls without modifying the browser, and system calls involve copying data, there is a risk that a benchmark may spend the majority of its time copying data in the kernel. Fortunately, our measurements indicate that this is not the case. Figure 1 shows the percentage of time spent in the kernel on Firefox when running the SPEC benchmarks. On average, each SPEC benchmark only spends 0.2% of its time in the kernel (the maximum is 1.2%); we conclude that the cost of Browsix-Wasm is negligible.

Benchmark	Native	Google Chrome	Mozilla Firefox
401.bzip2	370	864	730
429.mcf	221	180	184
433.milc	375	369	378
444.namd	271	369	373
445.gobmk	352	537	549
450.soplex	179	265	238
453.povray	110	275	229
458.sjeng	358	602	580
462.libquantum	330	444	385
464.h264ref	389	807	733
470.lbm	209	248	249
473.astar	299	474	408
482.sphinx3	381	834	713
641.leela	466	825	717
644.nab_s	2476	3639	3829
Slowdown:geomean	—	1.55x	1.45x
Slowdown:jmedian	—	1.53x	1.54x

Table 1: Detailed breakdown of SPEC CPU benchmarks execution times (of 5 runs) for native (Clang) and WebAssembly (Chrome and Firefox); all times are in seconds.

Measuring the Performance of WebAssembly Using SPEC

Finally, we are ready to consider the performance of the SPEC suite of benchmarks. Specifically, we used the C/C++ benchmarks from SPEC CPU2006 and SPEC CPU2017 (the new C/C++ benchmarks and the speed benchmarks). These benchmarks use system calls extensively and do not run without the support of Browsix-Wasm. We were forced to exclude four benchmarks that either failed to compile with Emscripten or allocated more memory than WebAssembly allows in the browser.

In Table 1 we show the absolute execution times of the SPEC benchmarks when running in Chrome, Firefox, and natively. All benchmarks are slower in WebAssembly, with the exception of 429.mcf and 433.milc, which actually run faster in the browser. Our ATC paper presents a theory of why this is the case. Nonetheless, most benchmarks are slower when compiled to WebAssembly: the median slowdown is nearly 1.5× in both Chrome and Firefox, which is considerably slower than the median slowdowns for PolybenchC. In our ATC paper, we also compare the performance of WebAssembly and JavaScript (asm.js) using these benchmarks, and confirm that WebAssembly is faster than JavaScript.

Explaining Why the SPEC Benchmarks Are Slower with WebAssembly

Using CPU performance counters, our ATC paper explores in detail why the SPEC benchmarks are so much slower when compiled to WebAssembly. We summarize a few observations below.

Register pressure. For each benchmark and browser, Figure 2 shows the ratio of the number of load instructions retired by WebAssembly over native code. On average, Chrome and Firefox retire 2.02× and 1.92× as many load instructions as native code, respectively. We find similar results for store instructions

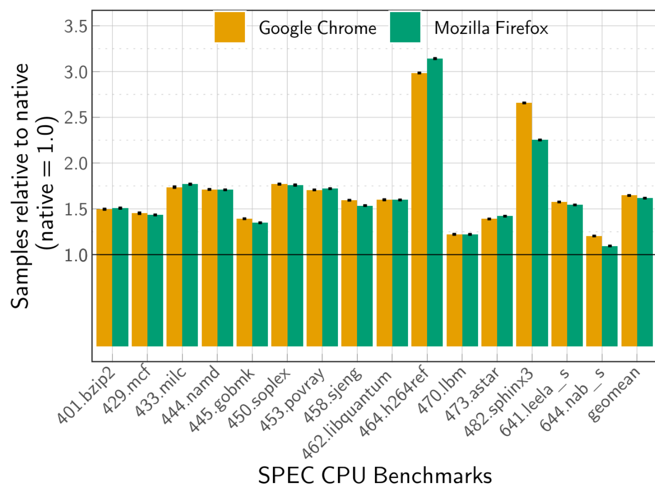


Figure 3: Ratio of the number of conditional branch instructions retired by WebAssembly over native code

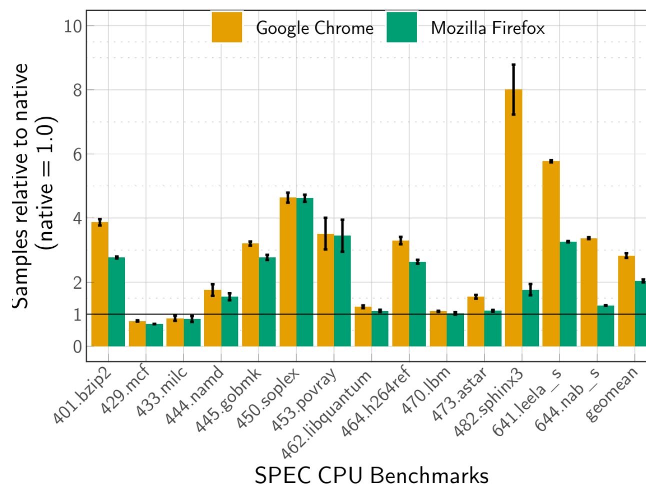


Figure 4: Ratio of the number of L1 instruction cache misses by WebAssembly over native code

retired. Our paper presents two reasons why this occurs. First, we find that Clang's register allocator is better than the register allocator in Chrome and Firefox. However, Chrome and Firefox have faster register allocators, which is an important tradeoff. Second, JavaScript implementations in Chrome and Firefox reserve a few registers for their own use, and these reserved registers are not available for WebAssembly either.

Extra branch instructions. Figure 3 shows the ratio of the number of conditional branch instructions retired by WebAssembly over native code. On average, both Chrome and Firefox retire 1.7× more conditional branches. We find similar results for the number of unconditional branches too. There are several reasons why WebAssembly produces more branches than native code, and some of them appear to be fundamental to the way the language is designed. For example, a WebAssembly implementation must dynamically ensure that programs do not overflow the operating system stack. Implementing this check requires a branch at the start of each function call. Similarly, WebAssembly's indirect function call instruction includes the expected function type. For safety, a WebAssembly implementation must dynamically ensure that the actual type of the function is the same as the expected type, which requires extra branch instructions for each indirect function call.

More cache misses. Due to the factors listed above, and several others, the native code produced by WebAssembly can be considerably larger than equivalent native code produced by Clang. This has several effects that we measured using CPU performance counters. For example, Figure 4 shows that WebAssembly suffers 2.83× and 2.04× more cache misses with the L1 instruction cache. Since the instruction cache miss rate is higher, the CPU requires more time to fetch and execute instructions, which we also measure in our paper.

Conclusion

We built Browsix-Wasm, a UNIX-compatible kernel that runs in a web page with no changes to web browsers. Browsix-Wasm supports multiple processes compiled to WebAssembly. Using Browsix-Wasm, we built a benchmarking framework for WebAssembly, which we used to conduct the first comprehensive performance analysis of WebAssembly using the SPEC CPU benchmark suite (both 2006 and 2017). This evaluation confirms that Wasm is faster than JavaScript. However, we found that WebAssembly can be significantly slower than native code. We investigated why this performance gap exists and provided guidance for future optimization efforts. Browsix-Wasm has been integrated into Browsix; both Browsix and Browsix-SPEC can be found at https://browsix.org.

Acknowledgments

Browsix-Wasm builds on earlier work by Powers, Vilk, and Berger (Powers and Berger are co-authors of this article). That work did not support WebAssembly and had performance issues that Browsix-Wasm addresses. This work was partially supported by NSF grants 1439008 and 1413985.

References

[1] 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 2018 USENIX Annual Technical Conference (USENIX ATC '19):* https://www.usenix.org/conference/atc19/presentation/jangda.

[2] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," 30th IEEE Symposium on Security and Privacy (Oakland '09), *Communications of the ACM*, vol. 53, no. 1, January 2010, pp. 91–99.

[3] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client Executables," 2010: https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf.

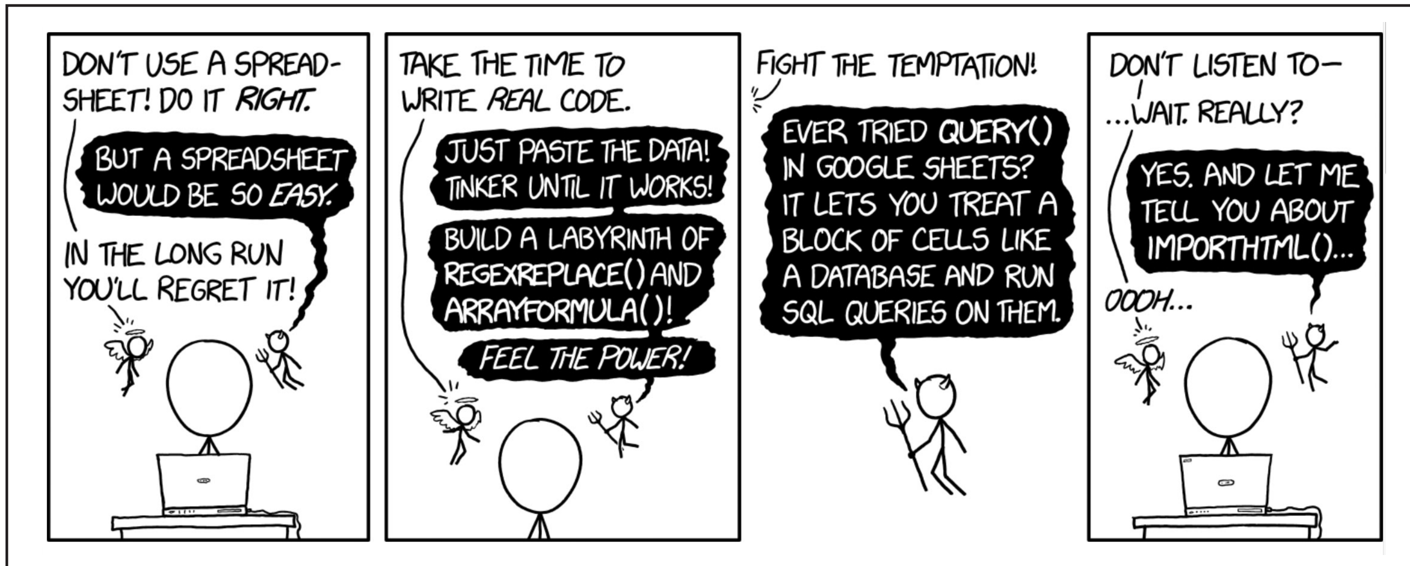
[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, ACM, 2017, pp. 185–200.

[5] PolyBenchC: The Polyhedral Benchmark Suite, 2012: http://web.cs.ucla.edu/~pouchet/software/polybench/.

[6] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*, ACM, 2011, pp. 301–312.

[7] WebAssembly JavaScript Interface, 2019: http://webassembly.github.io/spec/js-api/index.html.

XKCD **xkcd.com**



ENIGMA®

A USENIX CONFERENCE

SECURITY AND PRIVACY IDEAS THAT MATTER

Enigma centers on a single track of engaging talks covering a wide range of topics in security and privacy. Our goal is to clearly explain emerging threats and defenses in the growing intersection of society and technology, and to foster an intelligent and informed conversation within the community and the world. We view diversity as a key enabler for this goal and actively work to ensure that the Enigma community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and all talk media is available to the public after the conference.

PROGRAM CO-CHAIRS

Ben Adida
VotingWorks

Daniela Oliveira
University of Florida

The full program and registration will be available in November.

enigma.usenix.org

JAN 27–29, 2020

SAN FRANCISCO, CA, USA