Wast Wrong? Migrating WebAssembly Tests

Luna Phipps-Costin

April 18, 2023

1 Introduction

WebAssembly [Haas et al., 2017] is a programming language in which programs can be executed at near the speed of an equivalent C program. It is formally specified as well as specified in prose, and version 1.0.0 was formally verified in Isabelle/HOL [Watt et al., 2019]. The specification also includes a reference interpreter. In addition, WebAssembly is implemented in all of the major web browsers, as well as a number of implementations aimed at non-web environments, such as edge computing[1]. However, no implementation is verified. In addition, most implementations rely on complex optimizations and Just-In-Time compilation[2]. Further, many proposals have been integrated into the specification and/or implemented in major softwares. To address these issues, implementations rely on software testing. A test suite is provided as part of the specification, but its emphasis is more on clarifying the specification than detecting bugs. Without fail, implementations write additional tests.

However, these tests usually live in their own project's source tree, and may even be in incompatible formats.[3] Automatic test migration is the process of adapting tests from one program to another within a single domain. [Mariani et al., 2018, Behrang and Orso, 2019] We propose that performing test migration between these softwares may be beneficial. In particular, because WebAssembly is fully specified, with little ambiguity and extremely limited nondeterminism, end-to-end program tests often work as-is on every implementation, with divergences indicating a bug. While differential fuzzing (which many projects use) catches many bugs, the failures we find are primarily in bleeding-edge features and module-level edge cases that are likely not easily captured by fuzzing infrastructure.

2 wast: WebAssembly Scripts

WebAssembly has two fully-specified concrete syntaxes. One is a binary format optimized for space and load-time efficiency. The other, called the text format, maps cleanly to the binary format but is optimized for human readability. It consists of s-expressions with features such as proper identifiers and syntactic sugar. It also supports encoding binary modules using hexadecimal. The WebAssembly specification test suite is written in a light superset of the text language with additional syntax for multi-

[1] https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[2] https://v8.dev/docs/wasm-compilation-pipeline

[3] According to Andreas Rossberg, the spec testsuite should certainly be sufficient, and if it's not, the additional tests should be upstreamed to the spec. I don't know how practical or even desired that is.

ple modules and assertions. This language is called wast, or WebAssembly Scripts. [Rossberg, 2023] Programs written in wast can assert that functions return a particular value or trap, but can also assert that modules are malformed, fail to parse, or fail to typecheck, amongst others.

Because the specification includes tests in this format, and WebAssembly implementations are encouraged to test their conformance to this spec, many WebAssembly implementations not only support reading this format, but also write their own tests in it. This makes migrating tests particularly easy: not only do test programs conform to the WebAssembly spec, they also conform to a consistent abstract syntax. The Wast Wrong test suite collects tests written in this format from all supported implementations.

3 WebAssembly implementations

We use a given WebAssembly implementation for up to two purposes: we collect tests from its test suite, and we test the implementation. Each implementation might offer tests. have different difficulty levels for each engine. For example, while V8 can be downloaded from a system package manager, the test suite is not written the wast format: tests are written in a C++ macro system.[4] Similarly, JavaScriptCore is not distributed independently from its surrounding Web engine WebKit and thus is difficult to install, but the source code does contain wast tests.

Our harness supports testing the following implementations:

1. The reference interpreter is a supplement to the specification. There's a separate reference interpreter for each proposal. `https://github.com/WebAssembly/spec/tree/main/interpreter`

2. WasmTime is developed by the Bytecode Alliance, affiliated with Mozilla. Firefox currently uses its Javascript backend, to compile WebAssembly. However, they intend to eventually switch to WasmTime, so I won't test Firefox. `https://wasmtime.dev/`

3. V8 powers Google Chrome, Microsoft Edge, NodeJS, and many more, backed by Google. `https://v8.dev/`

4. SpiderMonkey powers Firefox. Firefox currently uses its Javascript compiler to compile WebAssembly. `https://spidermonkey.dev/`

5. Wizard is Ben Titzer's research implementation of WebAssembly. There are two implementations: a baseline compiler and an interpreter. We currently support only the interpreter. `https://github.com/titzer/wizard-engine`

Our test suite migrates tests from the following implementations:

1. WasmTime.

2. Wizard.

3. JavaScriptCore (JSC) powers Safari, Opera and others. A component of WebKit. `https://webkit.org/`

We expect that implementations conform to the spec test suite, and don't check that assertion.

[4] Ben Titzer is responsible for this and, thankfully, has personally apologized to me for it. His justification is that there was not yet a standard text format for WebAssembly when the V8 implementation was written.

4 Compiling wast

Unfortunately, not all WebAssembly implementations actually support reading the wast format. To support running these programs, we utilize off-the-shelf compilers from wast to JS or JSON.

wast2json, from the WebAssembly Binary Toolkit [Smith et al., 2023] splits each module in a wast files to a binary wasm file and encodes all assertions as JSON. We use wast2json to run tests with Wizard.

The reference interpreter from the specification supports compiling wast files to JavaScript. The resulting JavaScript includes WebAssembly, in binary format, hex-encoded in JavaScript, with assertions compiled to relevant JavaScript code. We use the interpreter to test the two Web implementations we support (v8 and SpiderMonkey).

Each compiler supports fewer proposed features than the programs we test. We simply fail to run these tests. This is unfortunate since testing proposed features is a key goal of this work, but increasing feature support for these compilers is a significant undertaking.

There was also a change in the naming of operators in the WebAssembly text (and thus wast) format in 2018. We were unable to find a compiler from the old syntax to the new one, but our harness is able to patch some tests that only used a few of the old operators.

5 Determining feature flags

In Wast Wrong, a migrated test can be seen as a pair of a test program and a configuration under which the implementation runs the program. Because WebAssembly is well specified with very little ambiguity, little configuration is necessary. However, WebAssembly is an evolving language, and many proposals exist to expand it with additional features. In addition, these proposals are often supported by implementations behind a feature flag long before standardization. Testing proposed features is a key goal of this work, because these features are likely to be less mature and less well-tested. In addition, fuzzing infrastructure may lag behind implementation in terms of feature support, making test migration especially useful.

In order to run tests utilizing or testing proposed features, we must enable the feature in the WebAssembly implementation using a feature flag. Enabling every feature for every test is an insufficient solution: many tests assert that modules fail to parse or are invalid, but an expanded language may make previously invalid modules valid. Fortunately, determining which tests require which features is usually trivial based on the path name. However, each new test suite may introduce new conventions by which to specify required features. Additionally, some tests do not indicate which features are required in their path. (See Evaluation.)

An earlier version of the Wast Wrong test harness attempted to identify features from that were unsupported by each given tool from the test path. However, this scales as O(n*m) where n is number of implementations and m is number of test sources, as each test source has a different convention to tag tests with required features. This is not sufficiently scalable, as we aim to be able to extend Wast Wrong in both directions in the future. Our solution is to separate feature use identification (using each test path) from unsupported feature detection (by implementation, using features from the former). This scales instead as O(n+m). As a note, identifying which features are supported is sometimes difficult. A good starting point is in the command-

line flags to a project, but some features may be enabled by default with no flag, so often times we've needed to search bug catalogs for positive indications of (lack of) feature support.

6 Testing projects without building from source

Building and modifying large software projects can be an undue burden. In Wast Wrong, we avoid building projects from source or running their test suites. Instead, we use off-the-shelf binaries to execute WebAssembly tests as compiled by our test harness (Section 4 / compiling wast).

However, it's impossible to avoid building some projects from source, as they don't distribute binaries compatible with my operating system, and my operating system does not package them. We still run these tools as black-box binaries for consistency, and to avoid modifying the programs / test infrastructure. A significant artifact of this work is a tool that builds and installs Wizard (and its dependency Virgil) and two additional versions of the reference interpreter from source. They are written as Nix packages [Dolstra et al., 2004], which allows them to be consistently, reproducibly built with their dependencies explicitly declared.

In addition, we use the Nixpkgs package archive to provide a Nix package that provides the three new packages along with the remaining WebAssembly implementations (except Wasmtime), and the two tools needed to run the harness. The result is that the entire harness can be built on any Linux x86_64 machine with no implicit dependencies except for Wasmtime. Wasmtime is not supported at this time because the version provided by Nixpkgs is greatly out of date, and newer versions require versions of Rust not easily obtained from Nixpkgs.

7 Evaluation

7.1 False positives

Our harness only logs failures which we are unable to automatically filter as spurious. The harness follows two rules to automatically determine if a test should be run and if the result is a genuine failure:

1. As described in Section feature flags, it skips each test that uses features known to be unsupported by a given implementation or by our harness

2. If the output of a test failure matches a known spurious messages, it considers it a spurious failure.

The harness currently describes only three known spurious messages, each describing a message that is raised on a differing but equivalent error message. Further, the number of messages is expected to scale only with the number of implementations, not with the number of tests or test sources.

We screened the test failures for results that were obviously spurious. We find twelve such false positives, and code them in six categories:

1. Test requires imports from a context. Some JSC tests expect to be provided values from the testing harness.

2. Test uses outdated syntax (Section 4 / Compiling wast)

3. Test uses features not indicated by path (Section 5 / Determing feature flags)

4. Test uses implementation-specific features. For example, wasmtime allows programs

Implementation	GC	Tail Call	Threads	Multi-Memory	Memory64	SIMD	Relaxed SIMD
Wasmtime	-	-	✔	✔	✔	✔	%
Wizard	✔	✔	✔	✔	-	✔	!
V8	!	✔	✔	-	!	✔	!
SpiderMonkey	-	-	✔	-	!	✔	!
Spec	!	✔	✔	!	!	✔	!

Table 1: WebAssembly feature support. - indicates features unsupported by the wasm implementation. ✔ indicates features supported by the implementation and harness. ! indicates features supported by the wasm implementation, but not the harness. % indicates partial support, which our harness considers lack of support.

to canonicalize NaNs, and has relevant tests, but no other implementations support this.

5. NaN comparison. Some tests assert that NaN must equal NaN. In wast, the semantics of this check are to assert that the two values are bit-wise equal, but WebAssembly allows for limited non-determinism within floating point values, making these tests often fail.

6. Inconsistent wast. For example, one Wasmtime test assumes it can refer to a module by stringified ‘‘`name`’’, but this does not work in all wast implementations.

The harness has affordances to easily add a list of known false-positives, and does not report these failures when run.

Note that identifying these false positives was a significant challenge. Following is a case study in detecting the NaN comparison false positives:

7.1.1 False positive: NaN Comparison

```
(module
  (func (export "bad-compare") ...
    (f32.min (f32.const NaN)
      (local.get $arg))))
(assert_return
  (invoke "bad-compare" (f32.const 42))
  (f32.const NaN))
```

WebAssembly allows for (limited) non-determinism in NaN (not-a-number) floating-point values. This test from Wizard assumes NaN's sign is preserved through floating point operations, which is not guaranteed by the spec. While the two NaN constants indicate positive NaN, the result of a float operation (such as f32.min) has an undefined sign. In V8, the result has a negative sign bit. The semantics of assert_return are such that the values are compared bitwise, leading this sign difference to lead to assertion failure. As a result, this test spuriously fails on V8 (along with 30 other floating point tests). Determining the exact meaning of the specification took quite a bit of time, and I believed this test to be a true failure even after reading the spec the first time.

7.2 Screened failures

However, we did not have enough time to positively confirm failures as bugs. This would require reporting failures as issues and waiting for

Implementation	JSC	Wasmtime	Wizard
Import from context	3	0	0
Outdated Syntax	2	1	0
Untagged feature	0	3	0
Implementation-specific	0	2	0
NaN comparison	0	0	31*
Inconsistent wast	0	1	0

Table 2: Definitive false positives. Tests that are excluded for causing false positives, by origin and reason for identification as a false positive. * marks 31 highly related tests.

Implementation	JSC	Wasmtime	Wizard
Wasmtime	0	0	0
Wizard	0	1	0
V8	0	0	1
SpiderMonkey	0	0	1
Spec	1	2	1
JSC	1	0	0

Table 3: Test failures by the wasm implementation and the source of test. Test failures have been screened as likely bugs.

confirmation from project maintainers. Due to the timeline of harness implementation and failure screening in addition to the expected timeline of responses, and in the effort to avoid bothering maintainers with too many spurious issues, we were forced to delay filing and confirming bugs until after the submission of this report.

While we are unable to confirm all failures as bugs, we have screened them as displaying odd behavior with no obvious out-of-band cause. We give quantitative results on the failures found, and explain one in detail. Note the odd result of a failure by JSC on a JSC test, despite the fact that we don't run JSC as a system under test. This corresponds to our case study, presented next.

7.3 Case Study: JSC Atomics

```
(module
  (func (export "cmpxchg") (result i64)
    (i64.store (i32.const 0)
      (i64.const 0xf0f0f0f0f0f0f0f0))
    (drop (i64.atomic.rmw8.cmpxchg_u
      (i32.const 0)
      (i64.const 0xf0f0f0f0f0f0f0f0)
      (i64.const 0x0000000000000011)))
    (i64.atomic.load (i32.const 0))))
(assert_return (invoke "cmpxchg")
  (i64.const 0xf0f0f0f0f0f0f0f0))
```

The above (minified) test fails on Wasmtime, V8, and SpiderMonkey. This test fails on some supported implementations but not others. In fact, the spec's informal and formal definitions of the instruction disagree. The test, formal spec, and reference interpreter agree that the exchange should not update the value. Following the informal spec, V8, Wasmtime, and SpiderMonkey update the value.

8 Future work

Reporting these bugs to maintainers will not only improve the implementations, it will also help better evaluate the false positive rate of the

harness by positively confirming failures as bugs or identifying new spurious failures.

Preliminary testing indicates that the test harness reports failures for the spec testsuite. Since we can assume all implementations pass the spec testsuite, we can use this as an oracle to better estimate the harness's false positive rate. However, the spec testsuite includes a greater number of tests than all implementation wast tests combined, and includes tests for every active proposal. Expanding the harness support to this number of tests, conventions, and features is a significant undertaking, and the result would (hopefully) find no new bugs.

The recall of the Wast Wrong testsuite could be ever expanded. More implementations could be added, both as systems under test, and as sources of tests. Some obvious test suites to be added are SpiderMonkey (although it has particular filesystem constraints), wasmi, wasm3, binaryen, wabt, wasmer, and wamr. For systems under test, besides testing many of the above, both Wizard's baseline compiler and JSC could be tested as well.

As noted in Section Compiling wast, some features are supported in implementations but not in the wast compilers. This could be improved. While the gaps in Wizard could only be attained by modifying the wast2json compiler, there is lower-hanging fruit. The harness for the Web implementations use the reference interpreter to compile wast files. While they currently use only one such interpreter, all proposals with implementations have a reference interpreter implementation that could in theory be used to compile the wast files. Each version of the reference interpreter would need to be updated in the Nix package to build from source, and feature detection would need to identify the appropriate binary to use for each test.

While wasm and wat files have no runtime oracle, the test harness could ensure that implementations can parse and validate these files.

Further test migration implementation could be done, such as writing a more complete translator from the old wast syntax to the new one, and providing values for imports in tests that expect them.

References

Farnaz Behrang and Alessandro Orso. Test Migration Between Mobile Apps with Similar Functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 54–65, San Diego, CA, USA, November 2019. IEEE. ISBN 978-1-72812-508-4. doi: 10.1109/ASE.2019.00016. URL `https://ieeexplore.ieee.org/document/8952387/`. 1

Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. 2004. 6

Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. 2017. 1

Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Augusto: exploiting popular functionalities for the generation of semantic GUI tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering*, pages 280–290, Gothenburg Sweden, May 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180162. URL `https://dl.acm.org/doi/10.1145/3180155.3180162`. 1

Andreas Rossberg. Webassembly scripts, 2023. URL https://github.com/WebAssembly/spec/tree/main/interpreter#scripts. Accessed 2023-04-18. 2

Ben Smith et al. Wabt: Webassembly binary toolkit, 2023. URL https://github.com/WebAssembly/wabt. Accessed 2023-04-18. 4

Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A Program Logic for First-Order Encapsulated WebAssembly. page 30 pages, 2019. doi: 10.4230/LIPIcs.ECOOP.2019.9. URL http://arxiv.org/abs/1811.03479. arXiv:1811.03479 [cs]. 1