November 28, 2026 · Development update

JS-Reactor: More JavaScript, less overhead

More code can now be virtualized, protected scripts run a little faster in several workloads, and we're testing JS-Reactor against considerably larger projects.

It's been a little over a week since the last website engine update on November 19. We've been working on JS-Reactor in the meantime, mainly on VM coverage, runtime performance and the kind of JavaScript that makes you wonder why the language even allows it in the first place.

One thing I don't want to do with JS-Reactor is make it faster by turning off protections or get a higher VM coverage number by breaking perfectly valid JavaScript. If the original script works, the protected script should work the same way. Sounds simple enough, until you actually have to implement it.

Runtime performance

The latest optimization round was tested against the preceding local engine build with the same workload and protection configuration. The improvements aren't massive, but they are measurable, and they come without a reduction in the tested VM coverage.

Protected execution · Node.js v26.13.0

Workload	Before	After	Paired change
Arithmetic / call	1.923 ms	1.868 ms	−4.9%
Branches / call	2.243 ms	2.149 ms	−4.9%
Strings / call	2.363 ms	2.358 ms	+1.9%
Cold process start	103.77 ms	103.14 ms	−0.3%

November 28 local release comparison, 12 paired samples per workload. Before and after are separate medians; the change column is the median of the individual paired percentage changes. Negative is faster. Cold start includes Node process startup and first execution. These are fixture results, not a universal speedup or a benchmark of the deployed Linux worker.

Compared with the earlier local reference build from the same development round, cold start was down 8.6%, arithmetic call time by 2.9% and branch call time by 6.6%. Strings were up 1.1%. So no, I'm not going to call the entire runtime 8.6% faster. It depends on what your script actually does.

What about output size?

At the end of October we reported an approximately 86% size reduction on our demo. I wanted to keep an eye on file size while expanding coverage, because nobody wants to download a couple of megabytes for a script that was a few kilobytes to begin with.

Protected output size · latest local round

Input fixture	Before	After	Paired change
Small	375 KB	375 KB	−0.2%
Medium	2.37 MB	2.34 MB	−1.0%
Large	8.32 MB	8.47 MB	+0.3%

12 paired builds per fixture; decimal KB and MB, rounded median output sizes. Paired percentage changes use the unrounded individual build sizes. Independently generated outputs naturally vary in size.

File size stayed in roughly the same range in these tests. The large fixture is still large, and there's more work to do here.

More JavaScript under the VM

We've expanded the set of functions JS-Reactor can virtualize, including supported immediately invoked function expressions and arrow functions. On a focused test containing six source functions, four are now virtualized instead of two. Existing compatibility behavior remains in place for code that cannot safely be virtualized.

There have also been improvements to the protection applied outside the VM. Code that needs to retain native JavaScript behavior still benefits from obfuscation. As usual, independently protected copies of the same script continue to vary in structure. We're keeping the implementation details of those protections private.

We still don't claim to virtualize every possible JavaScript program. Compatibility comes first, especially when working with complex modern JavaScript and third-party libraries.

We actually protected Babel

A little context from our November 15 testing: we took `@babel/standalone@7.28.5`, the entire 3.13 MB minified standalone compiler bundle, and ran it through JS-Reactor with maximum protection and the Chameleon profile.

The first protected file came out at 93.19 MB. Yes, 93 MB. Obviously this isn't a size anyone should be targeting for production web delivery; it was a stress test. What I wanted to know was whether Babel would still behave like Babel after protection.

The protected bundle ran under Node.js. The first workload checked the inventory of 13 presets and 123 plugins, plus 15 transformations. A second workload exercised all 13 presets, 11 plugin cases and three error cases. Both workloads produced byte-identical test results to the original bundle, across separate protected builds. That's a result I'm happy with, even though there's a lot of room to improve practical performance on a target this large.

Earlier compatibility stress test, not a new speed benchmark. Only the standalone UMD/CommonJS target and Chameleon were tested; this does not establish browser or other-profile compatibility for Babel.

Compatibility before everything

The current development build passed 165/165 checks in the normal compatibility matrix and 165/165 in the maximum-protection matrix. The workspace test run recorded 687 passing tests and one previously ignored test. We also checked independent protected demo builds and the focused new function cases across Chameleon, Orca and Cheetah.

That doesn't mean there will never be another bug. JavaScript has a seemingly endless supply of edge cases, and I'd rather find them in a test than have someone find them in a production script. If you have a reproducible script that behaves differently after protection, please let us know.

JS-Reactor Web IDE · Join the community on Discord
