Toward Automatically Detecting Cross-Engine Asymptotic Complexity Divergence in JavaScript

Minseok Choe
Korea University
Seoul, Republic of Korea
goro8pyo@korea.ac.kr

Abstract

The same JavaScript code can enter different asymptotic complexity classes in different engines, a phenomenon we call *cross-engine asymptotic complexity divergence*. Across V8, SpiderMonkey, and QuickJS, entering a `for..in` that breaks immediately is $O(1)$ in QuickJS and $O(N)$ in V8 and SpiderMonkey; a build-and-inspect string loop is linear in SpiderMonkey but quadratic in V8 and QuickJS. We propose automatic detection over parameterized programs $P(N)$: vary input size parameters and rank candidates by the spread of their estimated per-engine growth rates. Our goal is to find divergence cases and their triggering conditions. Engine-internal root-cause analysis and an empirical assessment of Algorithmic DoS exploitability are subsequent studies.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

JavaScript engines, performance testing, algorithmic complexity, differential testing

ACM Reference Format:
Minseok Choe. 2026. Toward Automatically Detecting Cross-Engine Asymptotic Complexity Divergence in JavaScript. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3832783.3844571

1 Introduction

The same JavaScript runs on V8 (Node.js, Chrome), SpiderMonkey (Firefox), and QuickJS (embedded and edge runtimes). On our 12-program calibration suite, their geometric-mean ratio is V8 1.00× : SpiderMonkey 1.35× : QuickJS 10.70×.

Developers may take such ratios as stable and apply the same general optimisation strategies across engines. The gaps, however, do not always hold. A small source-level change can instead put the same program in different complexity classes, creating an unbounded size-dependent gap: *cross-engine asymptotic complexity divergence*.

Automated performance testing usually considers one implementation, and cross-engine testing typically targets conformance rather than time (Section 2). We examine two patterns (Section 3) to motivate a size-varying search using comparative timing feedback from engines (Section 4). Our broader goal is to identify potential Algorithmic DoS vulnerabilities by detecting and characterising cross-engine asymptotic complexity divergences.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3844571

2 Background and Related Work

An empirical study of 98 fixed JavaScript performance issues [5] found that only 42.68% of the resulting optimisations improved performance consistently across all tested versions of V8 and SpiderMonkey. It reports variation in existing fixes, rather than searching for inputs that increase cross-engine performance differences.

Prior automated approaches evaluate one implementation at a time. JITProf [1] profiles a catalogue of JIT-unfriendly patterns in one engine. SlowFuzz [4] and PerfFuzz [2] maximise execution time for a fixed target. Jittery [6] compares optimisation levels of one compiler, finding 191 bugs. Our comparison axis is independent engines, not optimisation levels. Work that compares independent engines usually tests conformance rather than execution time [3].

3 Case Study

These two patterns were found through exploratory investigation, rather than by fuzzing. Each is a single program whose spread across engines is wider than the ordinary spread between those same engines.

How they are measured. We measure Node v25.2.0 (V8 14.x), SpiderMonkey 140.13.0, and QuickJS 2026-06-04 on an Apple M3 Max. Each cell in Tables 1 and 2 is a median wall-clock time (ms), followed by its ratio to that engine's geometric-mean baseline over 12 ordinary programs (1.00 : 1.35 : 10.70).

(1) Entering `for..in`. In V8 and SpiderMonkey, entering `for..in` over a large array costs $O(\text{length})$, even when the loop stops after the first key:

```
var a = []; for (var i = 0; i < 16000; i++) a.push(i);
var c = 0;
for (var r = 0; r < 2000; r++)
  { for (var k in a) { c = (c+1)|0; break; } }
```

Table 1: Entering `for..in` over a long array

loop form	V8	SM	QJS
`for (k in a) { break; }`	469 (125)	*1709 (337)*	0.13 (0.003)
`for (k in a) { }` all keys	611 (163)	*1723 (340)*	751 (19)
`for (v of a) { break; }` *(ctl)*	0.01 (0.003)	0.03 (0.006)	0.23 (0.006)

The controls show that the cost occurs when `for..in` starts. In V8 and SpiderMonkey, `for..in` is almost as slow when it stops after the first key as when it reads every key. Therefore, `break` does not avoid most of the cost. By contrast, `for..of` stays fast in all three engines as the array gets larger. Therefore, the cost does not come from `break` or from reading one array value. In QuickJS, stopping after the first key is fast, but reading every key costs 5,900× more. This result suggests that QuickJS does most of the work only when the loop requests more keys.

(2) Building and reading a string in one loop. Each iteration extends `s` and then reads it. Table 2 varies the read operation and the direction of string construction:

```
var s = "", c = 0;
for (var i = 0; i < 40000; i++)
  { s += "abcdefgh"; c = (c + OBSERVE)|0; }
```

Table 2: Building a long string and reading it

string build	OBSERVE	V8	SM	QJS
`s += x`	`s.length` *(ctl)*	0.24 (0.06)	0.11 (0.02)	2.00 (0.05)
`s += x`	`s.charCodeAt(0)`	*547 (146)*	0.92 (0.18)	105 (2.6)
`s += x`	`s[0] === "a"`	*548 (147)*	0.94 (0.19)	4.36 (0.11)
`s = s + x`	`s < "b"`	*638 (171)*	725 (143)	3.29 (0.08)
`s = x + s` *(ctl)*	`s < "b"`	0.38 (0.10)	0.15 (0.03)	2.08 (0.05)

The engines that become slow differ by row—SpiderMonkey and QuickJS each lose in one row and are fastest in another, while V8 is slow in all three. The controls narrow the triggering conditions. The `s.length` control shows that building string alone is cheap, and rows 2 and 3 differ only in an index versus a builtin method. The final control shows that the direction of string construction also matters for `s < "b"`. In row 4 both V8 and SpiderMonkey lose: `s < "b"` is decided by only the first character of `s`, yet QuickJS is 194× faster than either.

Scaling with input size. For each program, we varied the input size N from 1 to 10^6. We repeated one program call until the total time exceeded 20 ms, then divided the total time by the number of calls. This procedure makes $N = 1$ measurable. "—" marks a call that exceeded the 180 s timeout. Table 3 reports the results. Case (1) measures one `for..in` loop entry. Case (2) measures the full string-building loop.

Table 3: Time per call as the input size N grows

		1	10	10^2	10^3	10^4	10^5	10^6
(1)	V8	44 ns	74 ns	388 ns	4.1 µs	76.9 µs	1.53 ms	21.1 ms
	SM	84 ns	169 ns	1.1 µs	52.8 µs	759 µs	3.01 ms	18.1 ms
	QJS	83 ns	84 ns	85 ns	82 ns	84 ns	86 ns	82 ns
(2) `charCodeAt(0)`								
	V8	3 ns	291 ns	4.5 µs	124 µs	9.55 ms	3350 ms	—
	SM	3 ns	105 ns	1.6 µs	20.0 µs	209 µs	2.02 ms	25.6 ms
	QJS	88 ns	638 ns	4.9 µs	42.5 µs	6.78 ms	554 ms	—
(2) `s < "b"`								
	V8	4 ns	132 ns	8.2 µs	722 µs	70.9 ms	7474 ms	—
	SM	5 ns	59 ns	9.0 µs	917 µs	88.5 ms	8844 ms	—
	QJS	87 ns	658 ns	4.7 µs	39.9 µs	1.06 ms	11.98 ms	181 ms

QuickJS takes 82 ns to 86 ns to enter `for..in`, regardless of array length. Its time is therefore nearly constant. V8 and SpiderMonkey become slower as the array gets longer. We report each engine's growth rate as the exponent α in $t(N) \approx c \cdot N^{\alpha}$, obtained by a least-squares fit of $\log t$ against $\log N$ over the measured sizes; $\alpha \approx 1$ indicates linear growth and $\alpha \approx 0$ constant time. Across repeated runs, the fitted rates for V8 and SpiderMonkey are 1.26 and 1.37; QuickJS's rate is 0.02.

In the `charCodeAt(0)` row of case (2), SpiderMonkey's time grows approximately in proportion to N. V8 and QuickJS grow approximately with N^2. At $N = 10^5$, V8 takes 3.35 s, while SpiderMonkey takes 2.02 ms.

At $N = 10^6$, only one engine finishes each string case within three minutes, showing why size belongs to the candidate (Section 4).

4 Approach and Planned Evaluation

Building on the observations in Section 3, we propose the following fuzzing-based approach for finding such cases. It has not yet been evaluated; the reported cases were found through exploratory investigation.

Parameterized programs as candidates. The time gap between engines depends on the program *and* on the input size. If the fitness function rewards a large gap, the fuzzer can widen the gap by increasing the size alone. We therefore separate the two. A candidate is not one program but a *parameterized program* $P(N)$: a program with a size parameter N, such as a loop count, a property count, or a string length. The mutator changes the program. At execution time, N is instantiated with multiple values, and the program is run at each size.

Fitted growth rates as a score. For each engine e, we measure the running time at several sizes and fit $T_e(N) \approx c_e N^{k_e}$. Here c_e is a constant for that engine, and k_e is its estimated growth rate. The score of a candidate is $\Delta k = \max_e k_e - \min_e k_e$: the spread of the fitted growth rates across engines. A large constant gap gives a small Δk; a difference in complexity class gives a large one. Unlike the raw gap, Δk does not grow with N, so inflating the size no longer improves the score. Separating N from the program is also what makes the sweep, and hence the fit, possible.

Research questions. The planned evaluation asks two questions.
RQ1 (Effectiveness): Can the approach automatically detect cross-engine asymptotic complexity divergences? We check whether it rediscovers the two cases of Section 3, how many new divergences it finds, and whether each candidate's Δk reflects an actual divergence.
RQ2 (Robustness): How robust is the estimated growth-rate divergence to measurement noise, and the choice of input sizes? We vary repetitions, sampled input sizes, and the Δk threshold, and measure the stability of fitted growth rates and detection outcomes.

Future work. Detection establishes *that* a parameterized program diverges and *when*, not why. We will analyse engine internals to root-cause clustered triggering conditions, then test whether attacker-controlled valid inputs can turn those cases into Algorithmic DoS vulnerabilities [4], as in the path from pathological regular expressions to exploitable ReDoS.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597).

References

[1] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 357–368. doi:10.1145/2786805.2786831

[2] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. doi:10.1145/3213846.3213874

[3] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 13–24. doi:10.1109/ICSE43902.2021.00015

[4] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, 2155–2168. doi:10.1145/3133956.3134073

[5] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 61–72. doi:10.1145/2884781.2884829

[6] Zijian Yi, Cheng Ding, August Shi, and Milos Gligoric. 2026. Understanding and Finding JIT Compiler Performance Bugs. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (April 2026), 1237–1265. doi:10.1145/3798245

Received 2026-07-31; accepted 2026-08-25