

Toward Automatically Detecting Cross-Engine Asymptotic Complexity Divergence in JavaScript

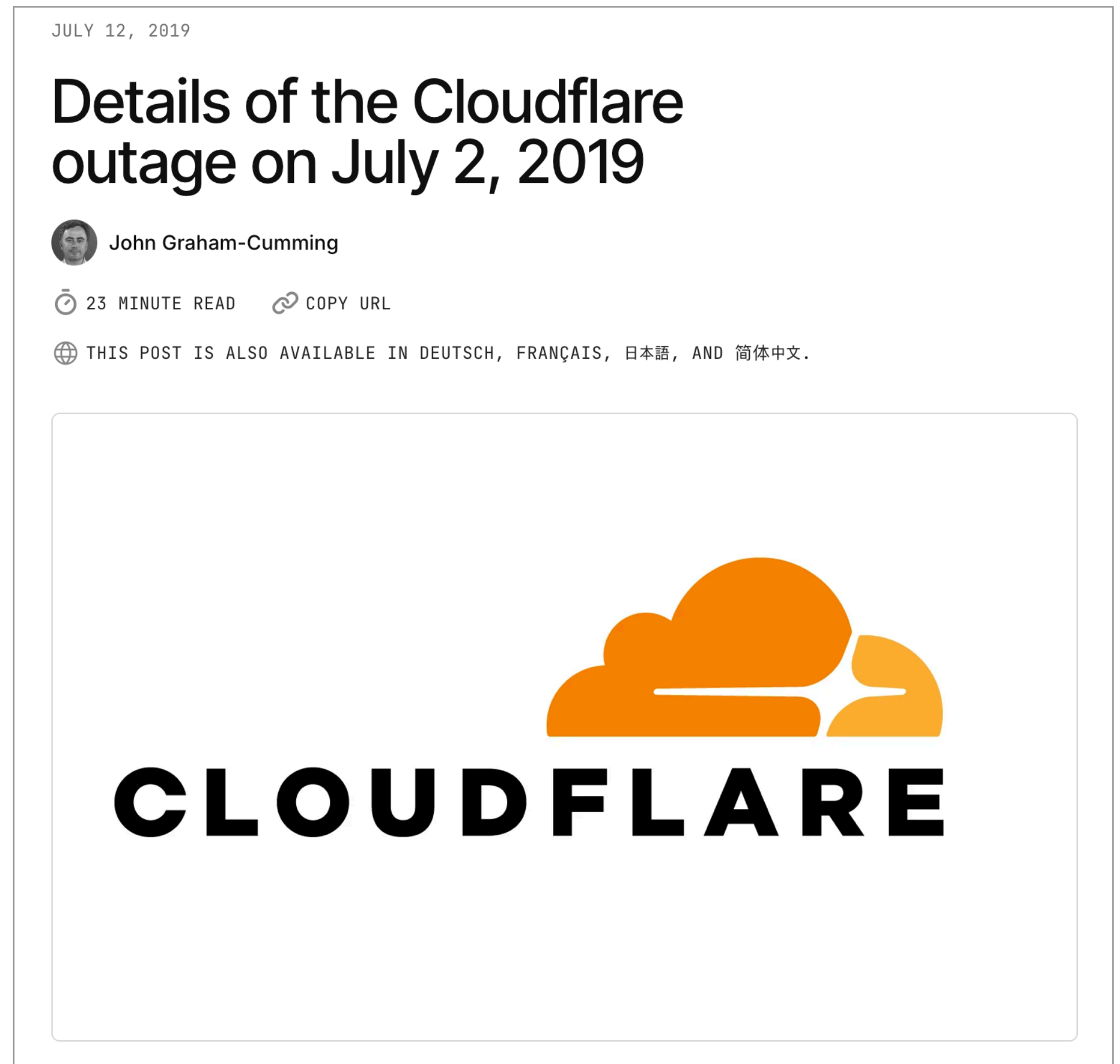
An Empirical Study of Algorithmic Complexity in JavaScript Engines

Minseok Choe, October 2026, ASE 2026 Student Research Competition



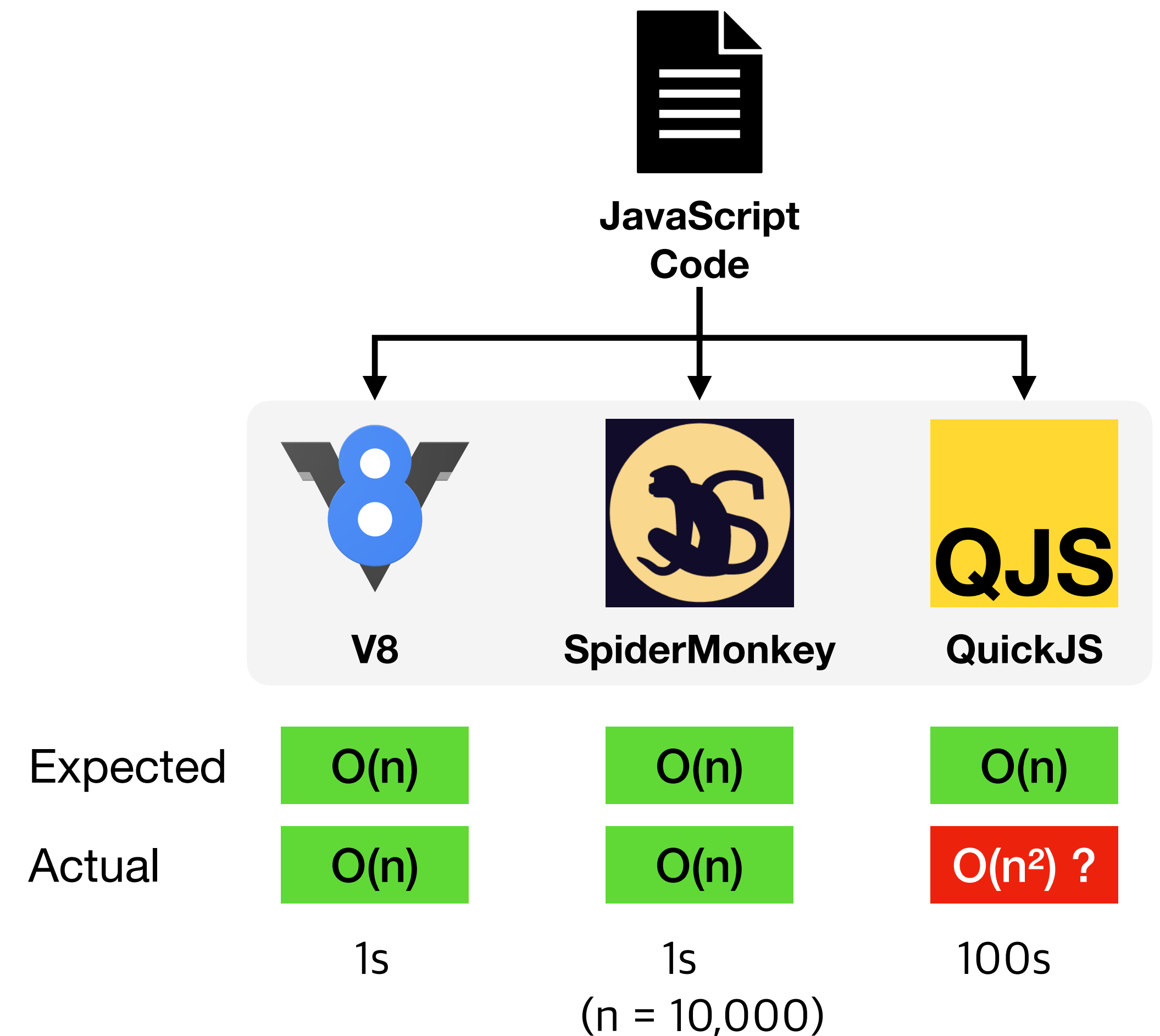
Motivation

- In 2019, an error in the Cloudflare firewall (WAF) stopped many services.
- A rule to find XSS attacks in URLs caused ReDoS.
 - The problem pattern (simplified): `.*(?:.*=.*`
 - The regex engine tries all possible matches(**backtracking**), so the number of steps increases very quickly.
- A slow algorithm can be an attack point of algorithmic DoS.



Motivation

- Developers expect the same complexity in all engines.
- One engine can be much slower when the input becomes large.
- Can an attacker use this to cause an algorithmic DoS?



Case Studies

Case 1: Loops with early breaks

```
const N = 16000;  
var array = [];  
for (var i = 0; i < N; i++) array.push(i);  
var effect = 0;  
  
for (var i = 0; i < BIG_NUMBER; i++) {  
    for (var _ in array) { effect += 1; }  
}
```

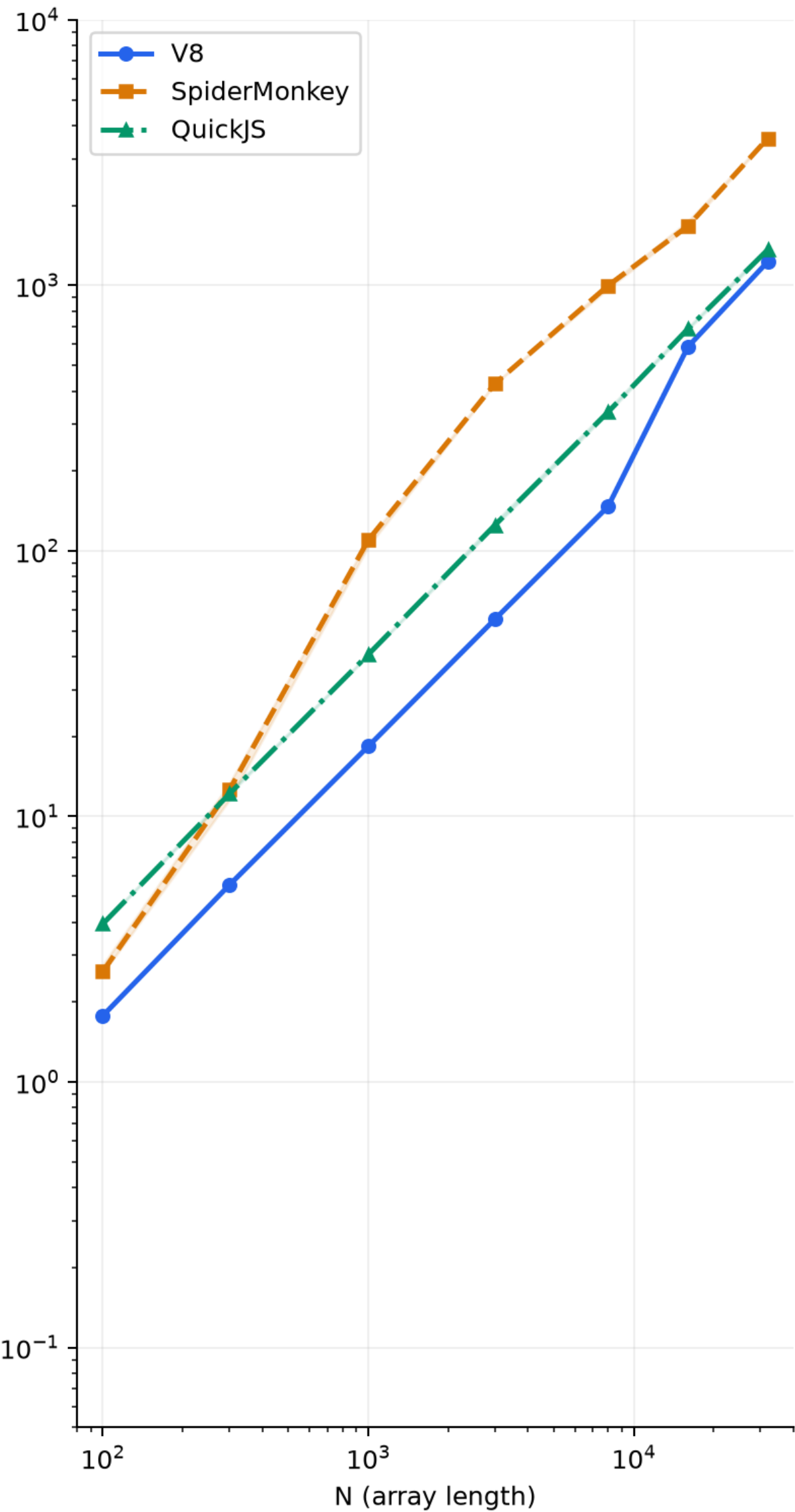
V8	SpiderMonkey	QuickJS
587ms	1672ms	685ms

Case 1: Loops with early breaks

```
const N = input();
var array = [];
for (var i = 0; i < N; i++) array.push(i);
var effect = 0;

for (var i = 0; i < BIG_NUMBER; i++) {
  for (var _ in array) { effect += 1; }
}
```

V8	SpiderMonkey	QuickJS
O(n)	O(n)	O(n)

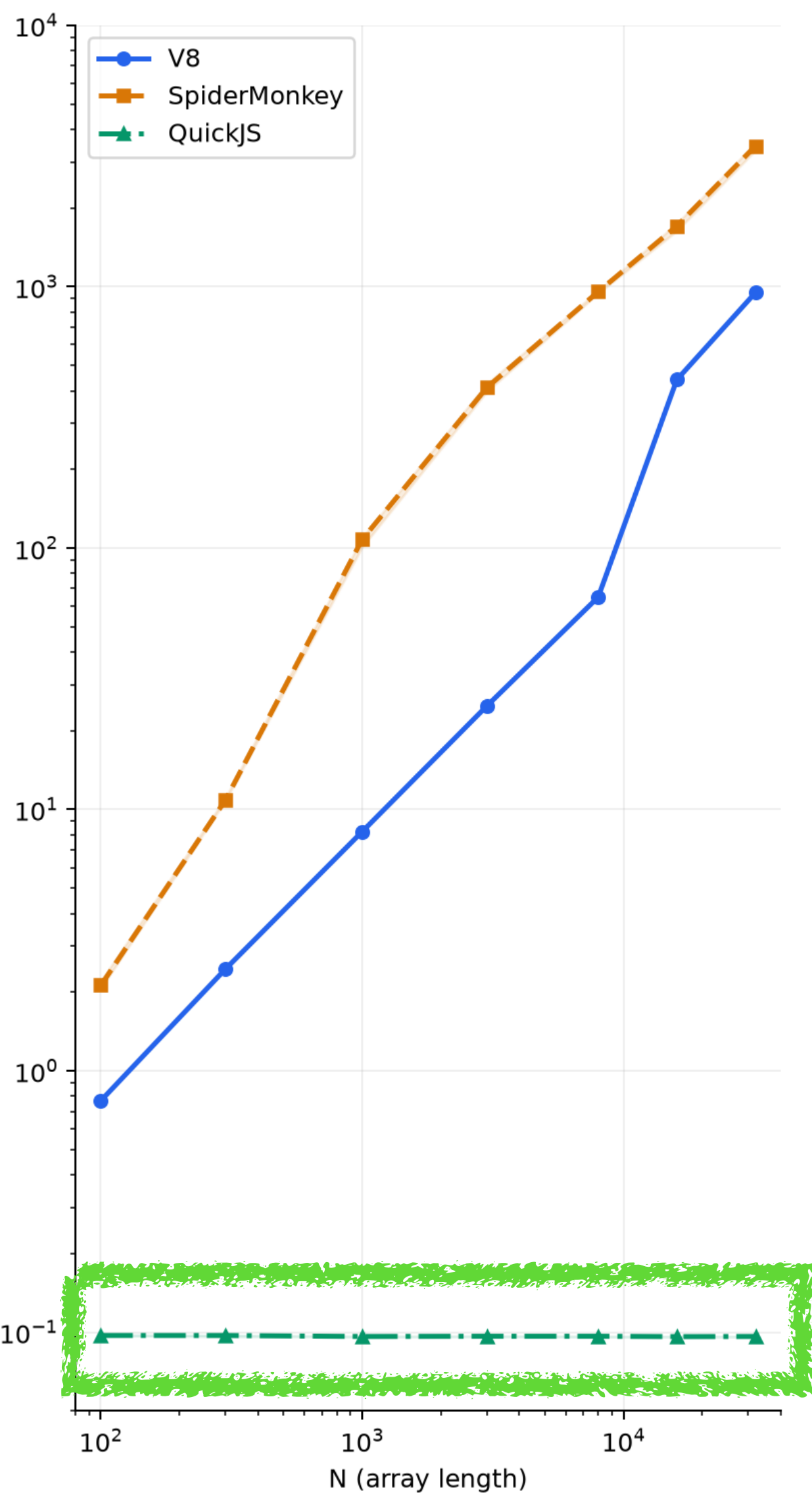


Case 1: Loops with early breaks

```
const N = input();
var array = [];
for (var i = 0; i < N; i++) array.push(i);
var effect = 0;

for (var i = 0; i < BIG_NUMBER; i++) {
  for (var _ in array) { effect += 1; break; }
}
```

Contains break;	V8	SpiderMonkey	QuickJS
No	$O(n)$	$O(n)$	$O(n)$
Yes	$O(n)$	$O(n)$	<u>$O(1)$</u>



Case 1: Loops with early breaks

Root Cause

- QuickJS has a **special path** for **fast array**:

QJS
QuickJS

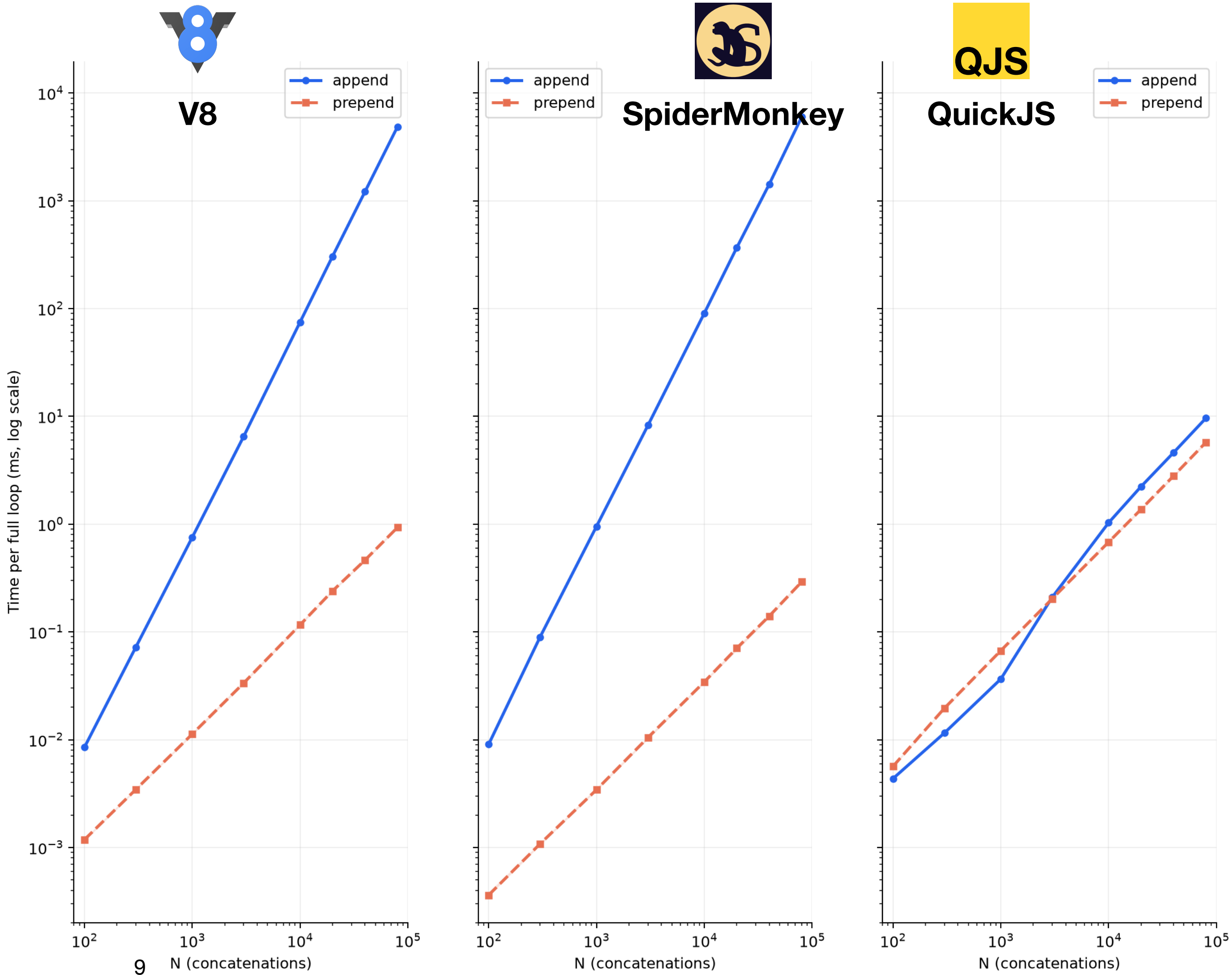
```
static JSValue build_for_in_iterator(JSContext *ctx, JSValue obj) { /* ... */
    if (p->fast_array) { /* ... */
        /* for fast arrays, we only store the number of elements */
        it->is_array = TRUE;
        it->atom_count = p->u.array.count;
    } else {
        normal_case:
        if (JS_GetOwnPropertyNamesInternal(ctx, &tab_atom, &tab_atom_count, p,
                                           JS_GPN_STRING_MASK | JS_GPN_SET_ENUM)) { /* ... */
            it->tab_atom = tab_atom;
            it->atom_count = tab_atom_count;
        }
        return enum_obj;
    }
}
```

- The same code can have a different asymptotic complexity in different engines.

Case 2: Building a long string

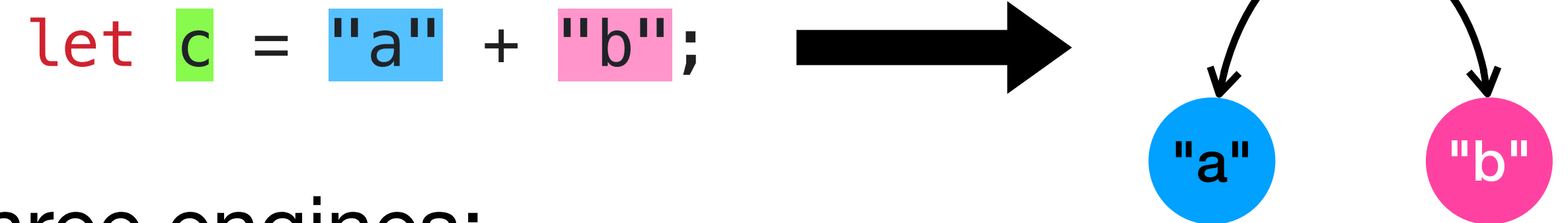
```
const N = input();
var str = "", effect = 0;
for (var i = 0; i < N; i++) {
  str = str + "abcdefgh";
  if (str < "b") effect++;
}
```

Direction	V8	SpiderMonkey	QuickJS
str = str + x (append)	O(n²)	O(n²)	O(n)
str = x + str (prepend)	O(n)	O(n)	O(n)



Case 2: Building a long string

Root Cause



- String concatenation is a **tree** in all three engines:

```
class ConsString extends String {  
  first: String;  
  second: String;  
}
```



V8

```
class JSRope : public JSString {  
  public:  
    JSString* leftChild()  
    JSString* rightChild()  
    // ...  
};
```



SpiderMonkey

```
typedef struct JSStringRope {  
  uint32_t len;  
  uint8_t is_wide_char;  
  uint8_t depth;  
  JSValue left;  
  JSValue right;  
} JSStringRope;
```



QuickJS

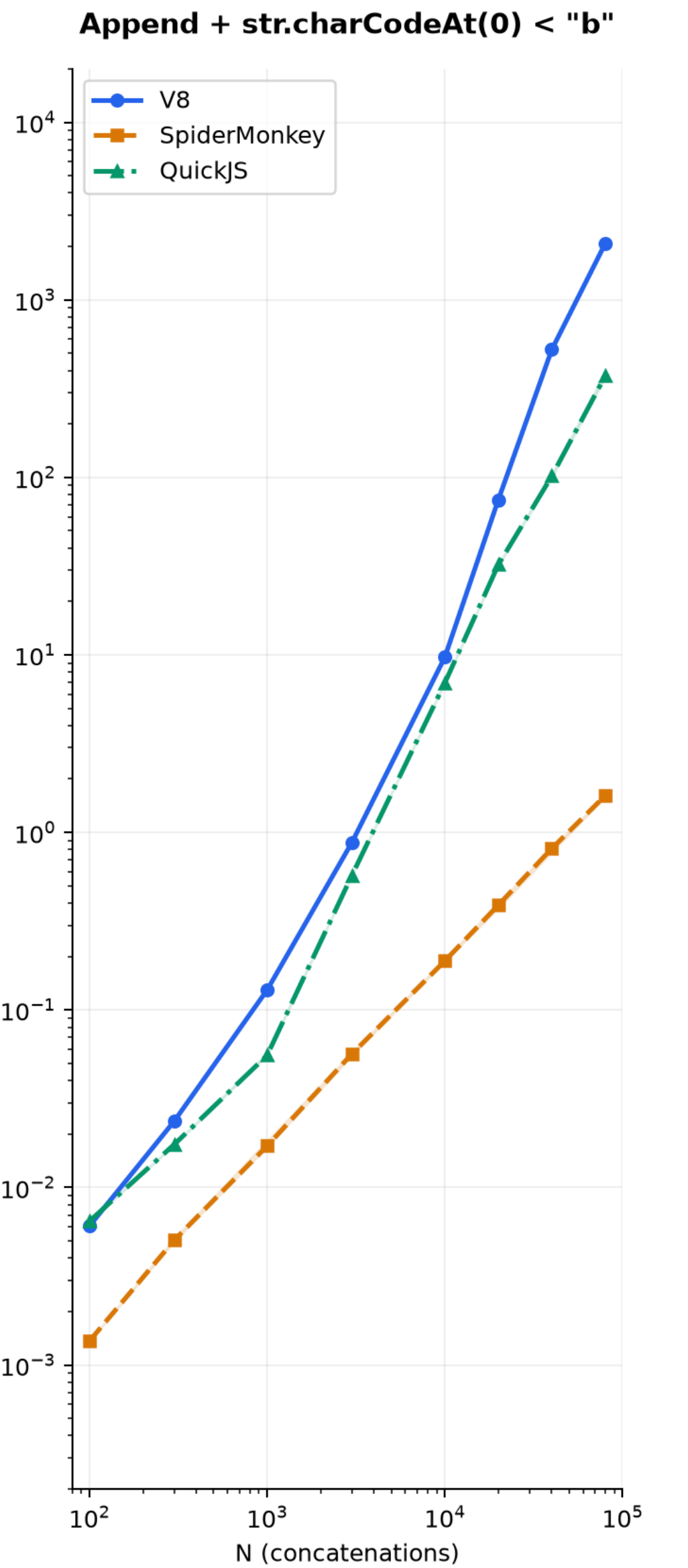
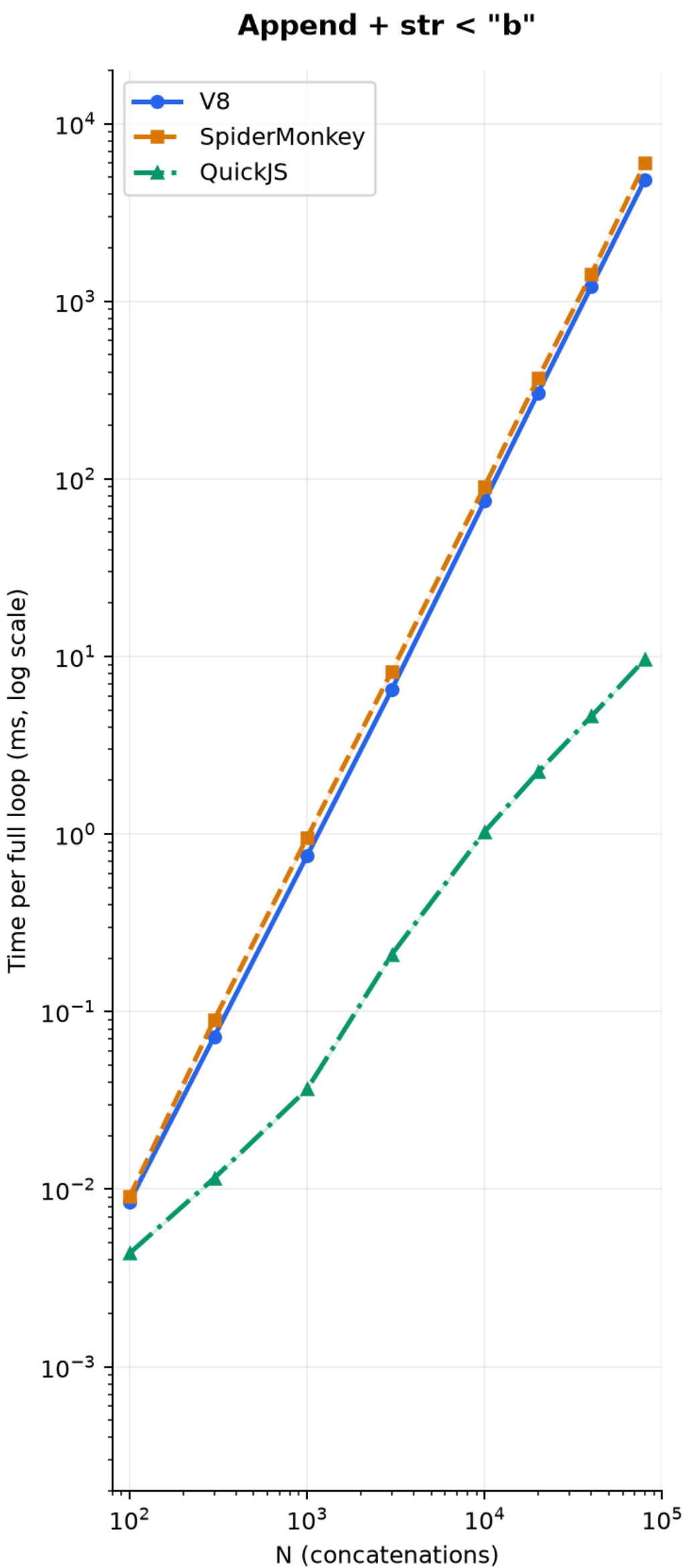
- Repeated appending (`str = str + x`) creates a **left-leaning** tree, making access to the **first (leftmost) character expensive**.
- QuickJS uses a **balanced** tree.

```
static JSValue js_rebalance_string_rope(JSContext *ctx, JSValueConst rope);
```

Case 3: Reading a long string

```
const N = input();
var str = "", effect = 0;
for (var i = 0; i < N; i++) {
  str = str + "abcdefgh";
  if (EXPR) effect++;
}
```

EXPR	V8	SpiderMonkey	QuickJS
str < "b"	$O(n^2)$	$O(n^2)$	$O(n)$
str.charCodeAt(0) < "b"	$O(n^2)$	<u>$O(n)$</u>	<u>$O(n^2)$</u>



Case 3: Reading a long string

Root Cause

- Before calling **built-in function**(charCodeAt), the string is **flattened** in all three engines.
- SpiderMonkey can reuse the existing buffer, avoiding repeated allocation.



SpiderMonkey

```
static bool CanReuseLeftmostBuffer(JSString* leftmostChild, /* ... */) {  
    if (!leftmostChild->isExtensible()) {  
        return false;  
    }  
    // ...  
    if (str.capacity() < wholeLength ||  
        str.hasTwoByteChars() != hasTwoByteChars) {  
        return false;  
    }  
    // ...  
    return true;  
}
```

- Other engines copy the full string each time: $O(n^2)$.

Conclusion

- JavaScript engines can have different asymptotic complexity for the same code.
 - Each engine uses different design choices in its implementation.
- A small change in code can cause asymptotic divergence between engines.
- Next step: find these divergences automatically