An Empirical Study of Lightweight JavaScript Engines

Meng Wu[1,2], Weixin Dong[3], Qiang Zhao[3], Zhizhong Pan[1,2], and Baojian Hua[1,2]*
[1]School of Software Engineering, University of Science and Technology of China, Hefei 230026, China
[2]Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China
[3]Guangdong OPPO Mobile Telecommunications Corp., Ltd. Shenzhen 210000, China
{wumeng21, sg513127}@mail.ustc.edu.cn, {dongweixin, zhaoqiang1}@oppo.com, bjhua@ustc.edu.cn*
* Corresponding author.

Abstract—Lightweight JavaScript engines have seen a considerable rise in recent years driven by the wide adoption of JavaScript programming for resource-constrained scenarios like edge computing and the Internet of Things. These resource-constrained scenarios present unique challenges for the lightweight JavaScript engines in terms of performance and memory usage. However, there is still a lack of comprehensive research that evaluates their reliability, especially concerning their support for modern ECMAScript standards and resilience.

In this study, we present, to the best of our knowledge, the *first* empirical analysis of lightweight JavaScript engines in four three domains: conformance to ECMAScript standards, performance evaluation, and resilience evaluation. We designed and implemented a software prototype called JASMIN, which we used to assess the four most widely adopted and representative lightweight JavaScript engines: QuickJS, JerryScript, Duktape, and MuJS. Our empirical results yielded valuable findings and insights, such as: 1) we proposed 3 root causes contributing to incompatibility with ECMAScript standards; and 2) we identified 3 issues related to executing obfuscated JavaScript code. 3) we investigated the performance of these lightweight JavaScript engines.

We believe that our study will serve as a useful reference for both JavaScript developers seeking optimal engine choices and for engine developers seeking to improve their products.

Keywords–Empirical study, Lightweight JavaScript engine, Software quality

I. INTRODUCTION

JavaScript [1], a widely used programming language initially developed for the web and browser, is increasingly being adopted in resource-constrained scenarios. This trend can be attributed largely to its technical advantages, such as its event-driven model and hot module replacement capabilities [2].

Specifically, in recent years, JavaScript has been successfully employed in a wide range of domains, including edge computing, micro-controllers, and Internet-of-Things (IoT), whose unique resource-constrained characteristics call for the development lightweight JavaScript engines. Consequently, several lightweight JavaScript engines such as QuickJS [3], JerryScript [4], Duktape [5], and MuJS [6], have been developed and are gaining increasing popularity. Going forward, the desire to program the Web-of-Things (WoT) [7] effectively is expected to accelerate the development and deployment of lightweight JavaScript engines.

Resource-constrained scenarios, characterized by limited computing resources, memory, power, and diverse hardware, pose significant challenges for implementing JavaScript virtual machines. Firstly, the size of the JavaScript engine must be optimized to fit constrained storage space. Secondly, depending on the application and device, JavaScript engines need to meet specific requirements such as low power consumption and fast startup. Thirdly, JavaScript engines must ensure compatibility with various device environments. These challenges require informed trade-offs in lightweight JavaScript engine implementation, considering factors like performance, supported language features, and robustness. Consequently, the quality assessment of lightweight JavaScript engines becomes more complex compared to desktop browser JavaScript engines. Notably, ECMAScript standard feature support and resilience are critical concerns for both developers and users of lightweight JavaScript engines.

Unfortunately, despite extensive investigations into desktop browser-side JavaScript engines, such as V8, SpiderMonkey, and Chakra, conducted in prior studies [8] [9] [10] [11], there is a lack of research on lightweight JavaScript engines. Specifically, to the best of our knowledge, there has been no comprehensive empirical investigation into lightweight JavaScript engines regarding their support for the ECMAScript standard and resilience. While some studies have compared lightweight JavaScript engines empirically [2], three key issues remain largely unexplored. First, a comprehensive evaluation dataset is still lacking. Previous studies have only proposed partial datasets to measure the ECMAScript standard or performance. Without a comprehensive evaluation dataset, the effectiveness of the testing process can be significantly compromised. Poorly designed or incomplete test cases may result in false positives or false negatives.

Second, the ECMAScript standard support of lightweight JavaScript engines has not been thoroughly investigated. It is critical that lightweight JavaScript engines support ECMAScript standard, as it determines to what extent JavaScript legacy code can be migrated to these engines. Unfortunately, existing studies [12] [13] have demonstrated that current engines such as Duktape [5] and Quad-wheel [14] lack support for the latest JavaScript standard, including essential built-in objects and methods like Date or Math. Furthermore, existing testing tools [15] [16] do not provide adequate support for the latest JavaScript standards.

Third, the resilience of lightweight JavaScript engines to obfuscated JavaScript code has not been thoroughly studied. It is crucial for lightweight JavaScript engines to demonstrate resilience against obfuscation, which is a commonly utilized technique to protect JavaScript source code and safeguard intellectual property by impeding reverse engineering efforts [17].

Therefore, in this study, we explore the following research questions that remain unanswered related to lightweight JavaScript engines. What is the evaluation dataset of lightweight JavaScript engines? To what degree do these lightweight JavaScript engines support the ECMAScript standard? What is the performance of lightweight JavaScript engines? How resilient are these engines in executing obfuscated JavaScript programs? The answers to these questions are vital as they provide valuable knowledge for JavaScript developers to effectively utilize these engines for secure and efficient code development. Additionally, engine developers can optimize their performance by acknowledging potential pitfalls.

Our work. In this paper, we aim to bridge this gap by presenting the first and most comprehensive empirical study of lightweight JavaScript engines. To conduct this study, we first designed and implemented a novel software tool prototype called JASMIN. Second, we selected and created 3 datasets to conduct this study: an ECMAScript standard dataset containing two sub-datasets, a performance benchmark suite including 14 tests, and an obfuscated JavaScript dataset with two sub-datasets. Third, we selected four popular and representative lightweight JavaScript engines: QuickJS [3], JerryScript [4], Duktape [5], and MuJS [6]. Finally, we perform an empirical study in terms of ECMAScript standard support, performance, and resilience.

We obtained important findings and insights from these empirical results, such as: 1) We investigated the ECMAScript standard support of these engines and present a taxonomy of scenarios that have failed to support ECMAScript standard features; 2) we investigated the ECMAScript standard support of these virtual machines and proposed 3 failure factors; 3) we proposed 3 issues by executing obfuscated JavaScript code in these engines and revealed the root causes. And our findings and suggestions have actionable implications for several audiences. Among others, they 1) help JavaScript developers to make more effective use of these lightweight JavaScript engines to develop effective and compliant JavaScript programs; and 2) help the engine developers further improve engines, by improving the ECMAScript standard support, performance, and resilience.

Contributions. To the best of our knowledge, this is the first and most comprehensive empirical study of lightweight JavaScript engines. To summarize, this work makes the following contributions:

- **Empirical study and tools.** We present the first empirical study of lightweight JavaScript engines with a novel software prototype JASMIN we created.
- **Datasets.** We created a comprehensive dataset for evaluating lightweight JavaScript engines.
- **Findings, insights, and suggestions.** We present interesting findings and insights, as well as suggestions, based on the empirical results.

Outline. The rest of this paper is organized as follows. Section II introduces the background for this work. Section III presents the approach we used to perform this study. Section IV presents the empirical results we obtained and answers to the research questions based on these results. Section V and VI discuss the implications of this work, and threats to validity, respectively. Section VII discusses the related work and Section VIII concludes.

II. Background

To be self-contained, this section provides necessary background knowledge for this work.

1. JavaScript and ECMAScript standard

JavaScript. JavaScript has maintained its status as one of the most popular programming languages for over two decades [18]. Initially developed by Netscape in 1995 to introduce dynamic functionality to web browsers [19], JavaScript has expanded its application domains to include servers [20], mobile applications, games, and desktop applications. JavaScript's technical advantages, such as hot module replacement and event-driven programming, make it particularly suitable for resource-constrained scenarios [12] [21], leveraging its robust ecosystem of libraries, frameworks, and toolkits.

The ECMAScript standard [22] serves as the official specification for JavaScript, defining its syntax, semantics, and APIs. Established by the European Computer Manufacturers Association (ECMA) and initially published as ECMA-262 in 1997 [23], this standard guides the implementation of JavaScript engines on various platforms. All major JavaScript engines, including V8, SpiderMonkey, and JavaScriptCore [24], have consistently supported the ECMAScript standard. With regular updates released annually, each new version of the ECMAScript standard imposes higher requirements on engine implementations.

2. Lightweight JavaScript Engines

Lightweight JavaScript engines have been specifically engineered for constrained-resource environments, such as embedded systems and IoT devices. These engines are designed with three key principles: efficient memory usage, a fast interpreter with minimal start-up time, and high embeddability for portability. Several widely adopted lightweight JavaScript engines are available today, including QuickJS, JerryScript, Duktape, and MuJS, each tailored to adapt to resource-constrained

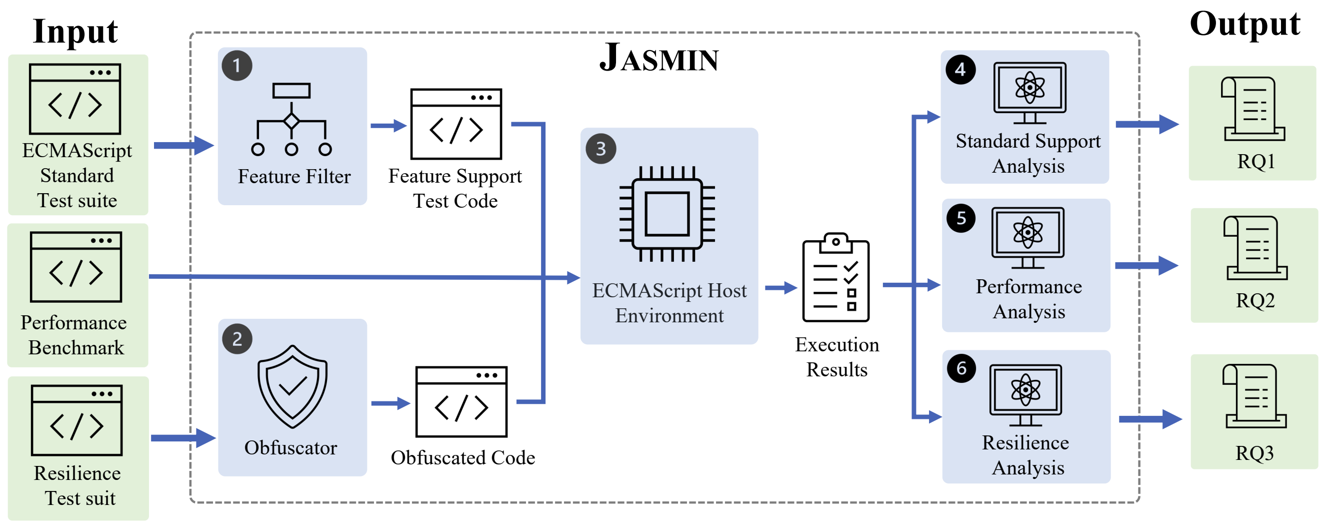

Figure 1: The Architecture of JASMIN.

environments. Firstly, these engines minimize memory consumption by having a small binary file size and using byte code instead of Just-in-Time (JIT) support. Secondly, they prioritize portability by requiring few system libraries, allowing them to operate on various platforms without customization.

Lightweight JavaScript engines typically consist of two main components: a parser and a virtual machine. The parser performs lexical analysis on the input JavaScript source code, generating an intermediary language, usually byte code customized for the engine. Subsequently, the virtual machine executes the sequential byte code instructions.

Many projects utilize lightweight JavaScript engines, such as game engines, static compilers, IoT frameworks, SQLite, monitors, and cross-platform UI development libraries [25] [26] [27]. These projects leverage lightweight JavaScript engines to provide efficient, fast, and low-resource alternatives to traditional JavaScript engines, enabling effective and efficient operation in resource-constrained environments.

III. Approach

This section presents our approach to conducting the empirical study. We have designed and implemented a software prototype JASMIN, to investigate research questions in an automated and scalable manner. We first present the design goals of JASMIN (Section III-A) and the architecture of JASMIN (Section III-B), then discuss the standard measurement module (Section III-C), the performance measurement module (Section III-D), the resilience measurement module (Section III-E), respectively.

1. Design Goals

The design goal of our JASMIN software tool is to achieve automaticity and scalability in conducting the empirical study of lightweight JavaScript engines. The study must be fully automated to enable the comprehensive and fully automatic evaluation of metrics for lightweight JavaScript engines. Human analysis is only necessary to supplement the automated analysis through manual code inspection. Additionally, the analysis should be scalable to accommodate various lightweight JavaScript engines, including potential future engines.

2. The Architecture

Based on these design goals, we present, in Figure 1, the architecture of JASMIN, consisting of seven key modules.

First, the feature filter module (❶) takes as input ECMAScript standard test suite, and screening for appropriate test cases based on specified ECMAScript features.

Second, the obfuscator module (❷) will take the resilience test suit as input and then obfuscate them and produce the obfuscated code. In the implementation, we use UglifyJS [28] as obfuscator because of its efficiency and practicality.

Third, the ECMAScript host environment module (❸) will normalize the host runtime environment for the different JavaScript engines, to ensure that each engine can execute in the same environment. In our implementation, we extended the ehost [29] tools to do this job.

Finally, the execution result analysis module processes the execution results and provides answers to the research questions (**RQs**) through three sub-modules: the standard compatibility analysis (❹), performance analysis (❺), and resilience analysis (❻). In the subsequent sections, we delve into the design and implementation of each module.

3. Standard Compatibility Analysis Module

The standard measurement module generates ECMAScript standard support results by running the input ECMAScript standard dataset on lightweight JavaScript engines according to user-supplied configuration.

It provides a comprehensive test of the lightweight JavaScript engine's support for ECMAScript standard with the following three steps: first, users should offer four configuration options including the JavaScript features to be included or excluded, the ECMAScript standard edition, and the output file path to the module. Second, the module identifies the sub-dataset that aligns with the configuration parameters. Then it proceeds to execute all language features included in the data set on the target lightweight engine. Finally, once the testing is complete, it processes the output results, which include the total number of test cases, the passed and failed number, as well as the type and number of supported JavaScript features and unsupported JavaScript features. The results from this measurement are used to answer **RQ1** (Section IV-A).

4. The Performance Analysis Module

To accommodate a resource-constrained environment, the lightweight JavaScript engines are designed to strike a balance between performance and memory usage. It is important to compare different lightweight JavaScript engines, examine their performance, and assist developers in optimizing JavaScript engines.

The performance measurement module employs a performance benchmark dataset to assess the efficiency of lightweight engines. Four criteria are utilized for testing purposes: binary file size, start-up time, execution time, and heap size. The execution steps of this module are as follows: first, it compiles the source code of the engine projects under a uniform compilation environment and tools to report the size of the binary executable files. Second, it calculates the cold start-up time by inserting a recording time function right after the engine initialization method is executed. Third, it uses the performance benchmark dataset as input, runs it on the target engine for 10 rounds, and outputs the average execution time and peak heap size. By generating reports based on these measurements, a root cause analysis is conducted to investigate the research question **RQ2** (Section IV-A).

5. The Resilience Analysis Module

We have incorporated the resilience module into the JASMIN to evaluate the resilience of lightweight JavaScript engines for obfuscated JavaScript datasets generated by the obfuscator.

The resilience module conducts testing by inputting the obfuscated JavaScript dataset and comparing the results with those of the same unobfuscated JavaScript dataset. The module compares the results to identify any deviations or discrepancies, and analyzes the root cause of any observed differences. It has been introduced to address **RQ3** (Section IV-A).

IV. EMPIRICAL RESULTS

This section presents our empirical results by answering the research questions.

1. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

RQ1: ECMAScript Standard Support. To what degree do these lightweight JavaScript engines support the ECMAScript standards? What are the failure factors leading to unsupported ECMAScript Standard features?

TABLE I: JavaScript engines leveraged by JASMIN.

Name	License	Github Stars	Implementation Language
QuickJS [3]	MIT	6.3k	C
JerryScript [4]	Apache-2.0	6.6k	C
Duktape [5]	MIT	5.6k	C
MuJS [6]	ISC	730	C

RQ2: Performance. What is the average execution time and memory footprint of these lightweight JavaScript engines? What are the root factors leading to the performance gap?

RQ3: Resilience. Are these lightweight JavaScript engines resilient to common program transformation such as obfuscation?

2. Evaluation Setup

All evaluations and measurements are performed on a server with one 20 physical Intel i7 core CPU and 64 GB RAM running Ubuntu 20.04.

3. Lightweight JavaScript Engines and Datasets

We first describe the lightweight JavaScript enginess selected, and datasets created in this study.

Lightweight JavaScript Engines Following prior studies [9], we selected lightweight engines according to the four criterions that the engine 1) has more than 500 stars on GitHub, representing the popularity, 2) is developed mainly in C (or C++), with the language's unit size accounting for 70%+ of the total project size, to mitigate the impact of the inherent language-related inconsistencies that may arise during the evaluation of performance, and obfuscation, 3) is still actively updated and maintained, and 4) is open source code, which facilitates us to investigate the root causes of the research results. As a result, We have selected 4 lightweight JavaScript engines: QuickJS, JerryScript, Duktape, and MuJS, as TABLE I presented, in our study.

Datasets. To conduct the empirical study, we need 3 datasets: ECMAScript standard test suite (**ES**), performance benchmark (**PB**) and obfuscated JavaScript test suite (**OF**), as shown in TABLE II. First, we created a dataset **ES** to measure the ECMAScript standard conformance with 105,476 test cases, by leveraging the Test262 [16], an official ECMAScript conformance test suite. We then divide this dataset into two sub-datasets: 1) **ES-5**, a dataset of the ECMAScript 5 edition with 11,725 test cases; and 2) **ES-next**, a dataset of the ECMAScript 2023 edition with 93,751 benchmarks. The division between the ECMAScript 5 edition and the ECMAScript 6 edition, along with its subsequent iterations, has been necessitated by significant dissimilarities between them. The ECMAScript 6 edition marked a major update to the ECMAScript language, introducing numerous new syntax, operators, primitives, and objects. It was a pivotal milestone in the evolution of ECMAScript.

TABLE II: Datasets used in our study.

Name	ES		PB	OF	
	ES-5	ES-next		OF-bef	OF-aft
#Cases	11,725	93,751	14	950	950

Starting from the ECMAScript 6 edition, a yearly version release strategy was implemented to facilitate the continuous and progressive development of the language over time [30].
Second, dataset **PB** is a performance benchmark. We selected a set of benchmark suites that came from SunSpider [31], Octane 2 [32], Kraken [33], and JetStream 2 [34], covering a variety of distinct workloads, as the performance benchmark suite. These benchmarks above have attained a high degree of authority and have been widely employed in previous studies on engine performance [10] [35]. Furthermore, we have removed test cases that are not suitable for lightweight implementation goals, such as 3D rendering and other computationally intensive tests. The final set of 14 cases encompasses a variety of features, such as garbage collection, object creation, object and property access, regular expressions, dates, and base64 conversion has been chosen to be the **PB**.
Finally, **OF** is an obfuscated JavaScript dataset including the unobfuscated JavaScript dataset as **OF-bef** and the obfuscated JavaScript dataset as **OF-aft**. We created the **OF** with the following two steps: 1) To guarantee the diversity of JavaScript tests, the tests must be random, comprehensive, and covering a broad range of ECMAScript language features. To achieve this, we have selected 950 JavaScript tests as **OF-bef** from the feature test files of these four lightweight JavaScript engines' code repository; and 2) To generate obfuscated JavaScript tests, we utilized a widely-used and highly effective obfuscation tool, known as UglifyJS [28] that has been used in prior works [36] [37] [38], to obfuscate the 950 normal JavaScript tests generated by step 1). This allowed us to achieve a corresponding set of 950 obfuscated JavaScript tests as **OF-aft**.

4. RQ1: ECMAScript Standards Support

To answer **RQ1** by investigating the ECMAScript standards conformance of lightweight JavaScript engines, we applied JASMIN to the dataset **ES** (including both **ES-5** and **ES-next** in TABLE II). The **ES-5** dataset is used to evaluate all four engines, whereas **ES-next** dataset was employed to assess the capabilities of QuickJS and JerryScript, as both claim to provide support for a wider range of modern features.
TABLE III presents the empirical results of ECMAScript Standard 5 edition support in the four JavaScript engines. The empirical results give interesting findings and insights. First, except for MuJS (with a success rate of 83.67%), the other 3 JavaScript engines have relatively high success rates (all beyond 95.00%). Second, QuickJS offers the best support for the ECMAScript standard 5 (with a success rate of 96.13%). We then explored the root causes leading to ECMAScript standard 5 edition support failures, and identified three key reasons, as presented by the last 3 columns in TABLE III: ECMAScript edition distinctions, Unicode edition discrepancies, and unsupported JavaScript features.
First, several JavaScript engines currently support ECMAScript 6+ edition, bug ignore the inconsistencies between the ECMAScript 6+ and ECMAScript 5 standards with respect to certain features. Such ignorances lead to compatibility issues when executing ECMAScript 5 testing. For example, both QuickJS and JerryScript ignore the semantics discrepancy of `ToLength()` function in ECMAScript 6 and in ECMAScript 5, leading to infinite execution of the following JavaScript program:

```
var objOne = { 0: true, 1: true, length: "
    Infinity" };
return Array.prototype.lastIndexOf.call(objOne,
    true) === -1;
```

Second, Unicode edition discrepancies caused standard conformance issues. For example, according to the Unicode definition, U+180E (Mongolian Vowel Separator) is no longer a space. However, in the present font manufacturer implementation, it has lost its space properties and instead gained the property of an invisible control character.
Third, JavaScript features in the internationalization APIs are not supported by all these four engines. TABLE V presents the empirical results of ECMAScript Standard 2023 edition support in QuickJS, JerryScript, and Duktape. Test the engines' standard support across four aspects: `AnnexB`, `built-ins`, `intl402`, and `language`.
The empirical results give interesting findings and insights. First, QuickJS has the best support of the ECMAScript standard 2023 edition with 76158 passed tests. Second, these four engines have poor support in `intl402` and `annexB`. Third, both QuickJS and JerryScript also exhibit poor support for 9 features shown in TABLE IV.

Summary: All four engines demonstrate strong support for ECMAScript 5 edition, with QuickJS and JerryScript also providing significant support for the latest ECMAScript standards. However, 9 features are currently poorly supported by these engines. As a result, developers must exercise caution when implementing these features in their programs to avoid compatibility issues.

5. RQ2: Performance

To answer **RQ2** by investigating the performance of lightweight JavaScript engines, we first compared the binary file size of the four JavaScript engines. Then calculated the cold start-up time. Finally, we applied JASMIN to the dataset **PB** (Section IV-C). Each JavaScript program is executed in 10 rounds to calculate an average running time and measure the maximum value of the heap size.
As shown in TABLE VI, except for QuickJS, the binary file sizes of the other three lightweight JavaScript engines typically do not exceed 500 KB, making it very suitable for

TABLE III: Test results on the dataset **ES-5**.

JavaScript Engine	Result		Failure Factors		
	#Failed	Success Rates	#ES Edition	#Unicode Edition	#Unsup Features
QuickJS	453	96.13%	296	12	145
JerryScript	457	96.10%	294	12	151
Duktape	487	95.85%	219	10	258
MuJS	1,915	83.67%	0	12	1,903

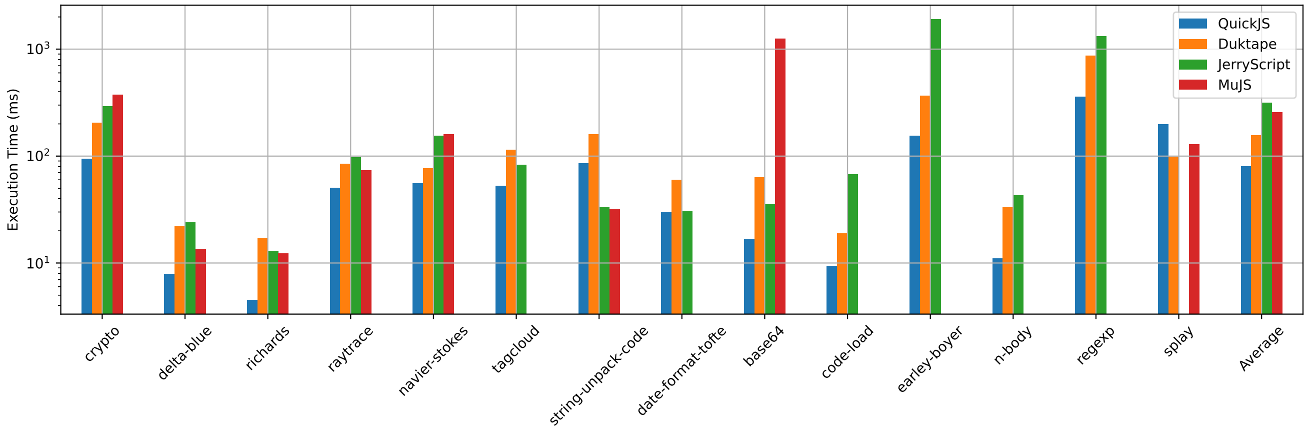

Figure 2: The Execution Time on the Dataset **PB**.

TABLE IV: Poorly supported features on the **ES-next**.

	QuickJS	JerryScript
Internationalization API	2%	2%
built-ins.Temporal	0%	0%
built-ins.Atomics.waitAsync	0%	0%
built-ins.Atomics.prototype	4%	0%
built-ins.ShadowRealm.prototype	0%	0%
language.waitAsync	0%	0%
annexB.built-ins.RegExp	56.4%	51.6%
language.module-code.top-level-await	49.9%	0.4%
built-ins.module-code.top-level-await	49.9%	0.4%

TABLE V: Passed test cases on the **ES-next**.

	ALL	QuickJS	JerryScript
annexB	1,360	1,306	1,019
built-ins	46,094	31,938	30,009
intl402	2,778	46	50
language	43,519	42,868	40,979
ALL	93,751	76,158	72,057

resource-limited application scenarios. It is worth mentioning that QuickJS has a binary file size of 1126 KB due to the inclusion of special libraries.

TABLE VII presents the start-up time of the four engines. The outcomes demonstrate that each of the four engines exhibits a remarkably swift start-up speed, which does not exceed 9 ms.

TABLE VIII and Fig.2 present the execution time results of running the dataset **PB**. First, on average, QuickJS (with an average execution time of 80.4 ms) outperforms the other three engines by more than 2 times in execution time, for the most part. Second, the time for executing the `base64` benchmark in the MuJS, averaging 1252.6 ms, surpasses that of the other three engines by more than 35 times.

TABLE IX and Fig.3 present the heap size results of running the dataset **PB**. First, JerryScript (with an average heap size of 106.3 KB) demonstrates a notable advantage over the other three engines in terms of memory usage. Second, the heap size required to perform the `base64` benchmark in the MuJS, with an average of 72,208 KB, exceeds that of the remaining three engines by a factor of 177.

We then explored the root causes, based on a manual inspection of the JavaScript engines' source code. This inspection revealed three key reasons: first, the design of byte code significantly impacts execution time and memory consumption. QuickJS prioritizes execution efficiency in its byte code design, while JerryScript focuses on memory conservation through techniques like compressed pointers and byte codes. However, this compressed byte code requires decompression during interpretation, resulting in slower execution times. On the other hand, Duktape's bytecode is not compressed to achieve relatively fast loading and access, but this leads to higher memory consumption.

Second, different garbage collection mechanisms will lead to different performance effects. QuickJS adopts the reference counting garbage collection mechanism, which allows for timely memory reclamation without the occurrence of a "stop

TABLE VI: Binary file size (KB).

QuickJS	Duktape	MuJS	JerryScript
1,126	341	352	440

TABLE VII: Start-up time (ms).

QuickJS	Duktape	MuJS	JerryScript
2	9	3	2

the world" scenario. In contrast, MuJS judges the necessity of garbage collection using mark-sweep before each instruction is executed to conserve memory as much as possible, resulting in slower execution times. JerryScript and Duktape provide both two mechanisms simultaneously.

Third, Duktape does not have adequate support for built-in objects and methods and some functions required by the standard, leading to longer program interpretation times which hinders its ability to optimize startup.

Summary: QuickJS performed the best in terms of execution time. JerryScript excelled in memory allocation. Duktape and MuJS closely followed. Overall, QuickJS had the best comprehensive performance.

6. RQ3: Resilience

To answer **RQ3** by evaluating the resilience of those lightweight JavaScript engines, we applied JASMIN to the dataset **OF**, consisting of obfuscated JavaScript programs, and TABLE X presents the comparison of the number of failures before and after obfuscation.

The empirical results give interesting findings and insights. First, JerryScript (with 0 new failure number of running **OF-aft**) demonstrated the highest resilience among the four engines compared, as the execution results remain unchanged before and after obfuscation. Second, QuickJS and Duktape (with 1 new failure number of running **OF-aft**) exhibited remarkable resilience by following JerryScript. Third, it seems like MuJS has experienced 3 failures when executing the **OF-aft** dataset, but for this particular test set, all of the failures were caused by the same type of reason.

We then explored the root causes by manually inspecting the JavaScript engine sources and comparing the pre-obfuscated and post-obfuscated JavaScript codes. This inspection unveiled three key reasons: first, the engine currently in use does not support the syntax features present in the post-obfuscated JavaScript codes. More precisely, the obfuscation tool supports the language features found in versions beyond ECMAScript 5. While obfuscating, new features may be employed to reorganize the code. However, these new features may not be supported by the engines, hence causing failures in executing the obfuscated JavaScript code. For example, Duktape does not support the feature of setting prototypes with `__proto__` in an object expression, which is present in ECMAScript 6. This is exemplified in the following code:

TABLE VIII: Execution time (ms).

	QuickJS	Duktape	MuJS	JerryS
crypto	93.9	205	372.8	292.4
delta-blue	7.9	22.3	13.5	24
richards	4.5	17.1	12.3	12.9
raytrace	50.4	84.9	73.7	97.4
navier-stokes	55.4	77.1	160.1	154.3
tagcloud	52.64	114.46	NA	82.6
string-unpack-code	85.6	159.3	32.1	33.1
date-format-tofte	29.7	60.2	NA	30.8
base64	16.8	62.9	**1252.6**	35.3
code-load	9.4	19	NA	67.1
earley-boyer	154.6	366.2	NA	1902.8
n-body	11	33.2	NA	42.8
regexp	357.5	863.5	NA	1319
splay	197.3	98.8	128.8	NA
Average	80.4	156.0	255.7	315.0

TABLE IX: Heap size (KB).

	QuickJS	Duktape	MuJS	JerryS
crypto	524	436	3204	56
delta-blue	712	3868	2964	32
richards	308	680	692	24
raytrace	396	516	976	36
navier-stokes	2580	2348	NA	20
tagcloud	3188	3184	NA	8
string-unpack-code	4280	1368	2204	8
date-format-tofte	220	1872	NA	16
base64	304	408	**72208**	12
code-load	5940	24756	NA	120
earley-boyer	13656	21876	NA	8
n-body	264	316	NA	12
regexp	3728	5168	NA	132
splay	NA	NA	NA	NA
Average	2776.9	5138.2	13708.0	106.3

```javascript
var n=[13.37],
n={a:n,length:n,
__proto__:[13.37,13.37,13.37]};
n.sort()
```

Executing `n.sort()` will throw an exception, as the object `n` and its prototype do not have the property `sort`.

Second, the design limitations of the engine hinder the successful execution of the obfuscated JavaScript code. For instance, MuJS imposes a limit of 100 on the maximum nested expression limit of the AST. In the obfuscation strategy, the semicolon following a JavaScript statement may be replaced with a comma to enhance the code's unreadability. Although this is a JavaScript feature where the statement is treated as an expression, it may lead to an AST nesting that exceeds the limit of 100. Consequently, when running these obfuscated codes, MuJS will throw an exception. The following code shows the regulations of MuJS on the limit of AST nesting:

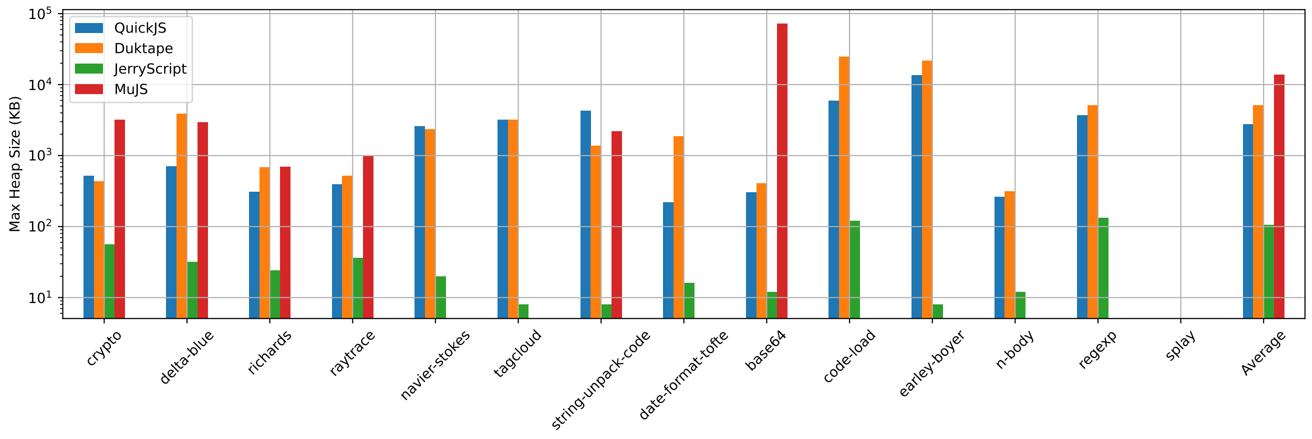

Figure 3: The Max Heap Size on the Dataset **PB**.

TABLE X: The comparison of the number of failures before and after obfuscation

	QuickJS	**Duktape**	**JerryScript**	**MuJS**
Before	0	75	97	306
After	1	76	97	309

```
// jsi.h: line 104
#define JS_ASTLIMIT 100  /* max nested
    expressions */
// jsparse.c: line 24
#define INCREC() if (++J->astdepth > JS_ASTLIMIT)
    jsP_error(J, "too␣much␣recursion")
```

Third, the engine's JavaScript tokenizer has some flaws, and as a result, the parsing of tokens in the obfuscated JavaScript code is failing. For example, if a method is called on a hexadecimal literal in QuickJS, it's necessary to enclose it in parentheses. Otherwise, the parsing will fail. The following code which is legal syntax in ECMAScript standard will trigger a parsing exception in QuickJS:

```
print(0xde0b6b3a7640080.toString());
```

Summary: All four lightweight engines, especially JerryScript, show good resilience in executing obfuscated code. However, the primary causes of failures are concentrated in three areas: incomplete feature support, design restrictions, and flawed lexical analyzers.

V. Implications

This work represents the first and most comprehensive empirical study of lightweight JavaScript engines, offering actionable implications for various audiences. This section discusses the implications of this research and highlights important directions for future studies.

For JavaScript Developers. Compared to traditional JavaScript engines, lightweight JavaScript engines emerged later and are currently experiencing rapid development. This study provides valuable insights into the standard support, performance, and flexibility. By gaining a deeper understanding of these key aspects, JavaScript developers can make informed decisions when selecting a suitable JavaScript engine and develop optimized and compatible code tailored to the specific features of their chosen engine.

Moreover, the provided prototype system, Jasmin, offers JavaScript developers the opportunity to test the latest JavaScript engine's standard support. This feature enables them to promptly apply the newest JavaScript features to their projects.

The findings of this research have significant implications for the JavaScript programming community, as they facilitate the development of more efficient and effective programming practices.

For engine developers. Developers of lightweight JavaScript engines face the challenging task of balancing performance and memory consumption to cater to resource-constrained scenarios. It is crucial for developers to optimize their engines to ensure efficient resource utilization while maintaining high-performance standards.

The results of this research provide valuable insights to developers of lightweight JavaScript engines regarding performance and memory optimization. By analyzing the data obtained from our prototype system, developers can identify areas in their code that require optimization and fine-tune their engines accordingly. Additionally, this research helps developers identify and fix code errors that may impact the resilience and code quality of their engines.

By following the optimization directions and addressing the code errors highlighted in this research, developers can make their engines well-suited to a wide range of use cases. This allows them to strike a balance between performance and memory consumption, enabling efficient resource utilization in resource-constrained scenarios.

For engine researchers. The empirical study offers valuable assistance to engine researchers in several ways. Firstly, the findings contribute to the development of standardized lightweight engine byte code, effectively balancing memory usage and performance optimization. Secondly, the results facilitate the formulation of interaction standards between lightweight JavaScript engines and emerging technologies like WebAssembly. This promotes more efficient and seamless integration of various technologies, enhancing overall functionality and performance of engines.

VI. Threats to Validity

As in any empirical study, there are threats to the validity of our work. We attempt to remove these threats where possible and mitigate the effect when removal is not possible.

Tools. In this work, we have used four lightweight JavaScript engines to conduct this study. Although these engines are widely used and thus represent state-of-the-art, there may be other engines available. Furthermore, new lightweight JavaScript engines might be developed in the future. Fortunately, the modular design of JASMIN makes it straightforward to testify to new engines. In the future, we plan to investigate other lightweight JavaScript engines when they are available.

Datasets. In this study, we used three different datasets, namely **ES**, **PB**, and **OF**. While **OF** was created using the tests included in those lightweight JavaScript engines' code repositories, there might be other datasets that can be used. Fortunately, the architecture of JASMIN is not specific to any particular dataset, which means that a new dataset can easily be added without any issues.

Errors in the Implementation. Most of our results are based on the JASMIN framework. Errors in the implementation could invalidate our findings. To mitigate this risk, we subjected all implementations to careful code reviews and tested them extensively.

VII. Related Work

There has been considerable research conducted on lightweight JavaScript engines; however, this paper represents a novel contribution to this field of research.

Studies on Lightweight JavaScript Engines. Extensive studies have been conducted on lightweight JavaScript engines. Sin et al. [2] proposed various IoT workloads to evaluate the performance and memory overhead of IoT systems, along with the assessment of several lightweight JavaScript frameworks. They also examined the effectiveness of multi-core systems for JavaScript frameworks. Kim et al. [12] analyzed the performance of three lightweight JavaScript engines, namely Quadwheel, Espruino, and Duktape. They further suggested optimization ideas for Duktape, implemented them, and achieved performance improvements and reduced memory footprint. Park et al. [39] discovered that a significant portion of heap memory is allocated to JavaScript source code, especially in lightweight JavaScript engines. To address this memory issue, they proposed dynamic code compression as an optimization technique to reduce source code memory consumption in JavaScript. Their work has been successfully integrated into the main branch of the Escargot engine [40]. However, they did not conduct large-scale empirical studies on lightweight JavaScript engines.

Empirical Studies of JavaScript Engines. Wang et al. [9] conducted the first empirical study on bugs in three mainstream JavaScript engines: V8, SpiderMonkey, and Chakra. Their study aimed to collect and classify bugs present in these engines. They found that the compiler and the DOM were the most bug-prone components in V8 and SpiderMonkey, respectively, and identified semantic bugs as the most common root cause. Park et al. [11] designed and implemented CRScope, an automated classification system for security and non-security bugs in JavaScript engines. Existing empirical studies on JavaScript engines primarily focus on large-scale mainstream engines, emphasizing bug research and analysis. However, there has been a lack of comprehensive assessments regarding ECMAScript standard support, performance, resilience, and code quality specifically for lightweight JavaScript engines.

VIII. Conclusion

We presented the first study of lightweight JavaScript engines. By designing and implementing a software prototype JASMIN, we evaluated four lightweight JavaScript engines in terms of ECMAScript standard support, performance, resilience and code quality. We found the root causes for the lack of ECMAScript standard support and poor performance. Furthermore, we explored and analyzed the root causes for the resilience of obfuscated JavaScript code, and evaluated the code quality of lightweight JavaScript engines. Our recommendations can benefit both JavaScript developers and engine developers, and can help to create a healthier ecosystem for lightweight JavaScript engines.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is supported by OPPO Research Fund.

References

[1] C. Severance, "Javascript: Designing a language in 10 days," *Computer*, vol. 45, no. 2, pp. 7–8, Feb. 2012.

[2] D. Sin and D. Shin, "Performance and resource analysis on the javascript runtime for iot devices," in *Computational Science and Its Applications – ICCSA 2016*, ser. Lecture Notes in Computer Science, O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds. Springer International Publishing, pp. 602–609.

[3] Quickjs javascript engine. [Online]. Available: https://bellard.org/quickjs/

[4] Jerryscript. [Online]. Available: https://jerryscript.net/

[5] Duktape. [Online]. Available: https://duktape.org/

[6] Mujs. [Online]. Available: https://mujs.com/

[7] L. Sciullo, L. Gigli, F. Montori, A. Trotta, and M. D. Felice, "A survey on the web of things," vol. 10, pp. 47 570–47 596.

[8] G. Dot, A. Martinez, and A. Gonzalez, "Analysis and optimization of engines for dynamically typed languages," in *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, pp. 41–48. [Online]. Available: http://ieeexplore.ieee.org/document/7379832/

[9] Z. Wang, D. Bu, N. Wang, S. Yu, S. Gou, and A. Sun, "An empirical study on bugs in javascript engines," vol. 155, p. 107105. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584922002142

[10] H.-K. Choi and J. Lee, "Optimizing constant value generation in just-in-time compiler for 64-bit javascript engine," vol. 43, no. 1, pp. 34–39. [Online]. Available: https://koreascience.kr/article/JAKO201605555187405.page

[11] S. Park, D. Kim, and S. Son, "An empirical study of prioritizing javascript engine crashes via machine learning," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19. Association for Computing Machinery, pp. 646–657. [Online]. Available: https://doi.org/10.1145/3321705.3329840

[12] G. Keramidas, N. Voros, and M. Hübner, Eds., *Components and Services for IoT Platforms*. Springer International Publishing. [Online]. Available: http://link.springer.com/10.1007/978-3-319-42304-3

[13] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+1-version differential testing of both javascript engines and specification," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, pp. 13–24. [Online]. Available: https://ieeexplore.ieee.org/document/9402086/

[14] "Google code archive - long-term storage for google code project hosting." https://code.google.com/archive/p/quad-wheel/.

[15] "Test262 report," https://test262.report/.

[16] "Test262-harness," https://www.npmjs.com/package/test262-harness, Jun. 2022.

[17] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, pp. 9–16. [Online]. Available: http://ieeexplore.ieee.org/document/6461002/

[18] "Tiobe index," https://www.tiobe.com/tiobe-index/.

[19] A. Wirfs-Brock and B. Eich, "Javascript: The first 20 years," vol. 4, pp. 1–189. [Online]. Available: https://dl.acm.org/doi/10.1145/3386327

[20] T.-L. Tseng, S.-H. Hung, and C.-H. Tu, "Migratom.js: A javascript migration framework for distributed web computing and mobile devices," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, pp. 798–801. [Online]. Available: https://dl.acm.org/doi/10.1145/2695664.2695987

[21] "Javascript empowered internet of things — ieee conference publication — ieee xplore," https://ieeexplore.ieee.org/abstract/document/7724687.

[22] E. C. M. Association (ECMA), "Ecmascript language specication." [Online]. Available: https://cir.nii.ac.jp/crid/1573105975500709504

[23] C. Dittamo, V. Gervasi, E. Boerger, and A. Cisternino, *A Formal Specification of the Semantics of ECMAScript*. Università di Pisa. [Online]. Available: http://compass2.di.unipi.it/TR/Files/TR-11-02.pdf.gz

[24] Javascriptcore — apple developer documentation. [Online]. Available: https://developer.apple.com/documentation/javascriptcore

[25] "Iot.js," https://iotjs.net/.

[26] "Introducing javascript* runtime for zephyr™ os - zephyr project," https://www.zephyrproject.org/introducing-javascript-runtime-for-zephyr-os/.

[27] https://wiki.duktape.org/projectsusingduktape.

[28] Uglifyjs — javascript parser, compressor, minifier written in js. [Online]. Available: https://lisperator.net/uglifyjs/

[29] Execute ecmascript code uniformly across any ecmascript host environment. [Online]. Available: https://github.com/bterlson/eshost

[30] J. D. Isaacks, *Get Programming with JavaScript Next: New Features of ECMAScript 2015, 2016, and Beyond*. Simon and Schuster.

[31] Sunspider 1.0 javascript benchmark. [Online]. Available: http://proofcafe.org/jsx-bench/js/sunspider.html

[32] Octane 2.0 javascript benchmark. [Online]. Available: https://chromium.github.io/octane/

[33] Kraken. [Online]. Available: https://wiki.mozilla.org/Kraken

[34] Jetstream 2. [Online]. Available: https://browserbench.org/JetStream2.0/

[35] J. Radhakrishnan, "Hardware dependency and performance of javascript engines used in popular browsers," in *2015 International Conference on Control Communication & Computing India (ICCC)*, pp. 681–684.

[36] G. S. Ponomarenko and P. G. Klyucharev, "On improvements of robustness of obfuscated javascript code detection." [Online]. Available: https://doi.org/10.1007/s11416-022-00450-1

[37] S. Rauti and V. Leppanen, "A comparison of online javascript obfuscators," in *2018 International Conference on Software Security and Assurance (ICSSA)*. IEEE, pp. 7–12. [Online]. Available: https://ieeexplore.ieee.org/document/9092263/

[38] B. Bertholon, S. Varrette, and P. Bouvry, "Comparison of multi-objective optimization algorithms for the jshadobf javascript obfuscator," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pp. 489–496.

[39] H. Park, S. Kim, and B. Bae, "Dynamic code compression for javascript engine," *Software: Practice and Experience*, vol. 53, no. 5, pp. 1196–1217, May 2023.

[40] "Escargot," Samsung, Apr. 2023.