An Empirical Study of Lightweight JavaScript Virtual Machines

Meng Wu Baojian Hua* Zhizhong Pan
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{wumeng21, sg513127}@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—The lightweight JavaScript virtual machines have become increasingly ubiquitous in recent years, because JavaScript programming is widely used on scenarios that have limited resources, such as mobile applications, edge computing, and the Internet of Things. These resource-constrained scenarios present unique challenges for the lightweight JavaScript virtual machines in terms of performance and memory usage. However, little is known about whether these lightweight JavaScript virtual machines can meet the specific requirements of resource-constrained devices.**

In this paper, we present, to the best of our knowledge, the *first* and most *comprehensive* empirical study of lightweight JavaScript virtual machines in five key areas: creation of lightweight JavaScript engine evaluation datasets, ECMAScript standard support, performance evaluation, resilience analysis, and code quality. We first designed and implemented a software prototype JASMIN, and then used it to evaluate the four most popular and representative lightweight JavaScript virtual machines: QuickJS, JerryScript, Duktape, and MuJS. We obtained important findings and insights from empirical results, such as: 1) we proposed 3 root causes leading to unsupported JavaScript features; 2) we identified 3 issues related to executing obfuscated code; and 3) we investigated the code quality of these lightweight JavaScript virtual machines. We suggest that: 1) JavaScript developers should strengthen their programs by considering the different performance of lightweight JavaScript virtual machines; and 2) virtual machine developers should enhance lightweight JavaScript virtual machines in terms of ECMAScript standard support, performance, and completeness. We believe that our study will provide valuable guidelines for lightweight JavaScript engine studies, benefiting both JavaScript developers and engine developers.

***Index Terms*—Empirical study, Lightweight JavaScript virtual machines, Software quality**

I. Introduction

JavaScript [1], a language originally designed for Web and browser, is gaining popularity in resource-constrained domains, due to its technical advantages of event-driven model and hot module replacement [2]. Specifically, in recent years, JavaScript has been successfully used in a large spectrum of domains such as mobile systems, real-time systems, games, edge computing, micro-controllers, and Internet-of-Things (IoT), whose unique resource-constrained characteristics call for the development lightweight JavaScript virtual machines. In response, several lightweight JavaScript virtual machines (*e.g.*, QuickJS [3], JerryScript [4], Duktape [5], and MuJS [6]), have been developed and are gaining more popularity. For example, the GitHub stars of QuickJS [3], one of the most widely used JavaScript virtual machine, have increased significantly from 1,827 in 2021 to 5,840 in 2023 [7]. In the future, a desire to program the Web-of-Things (WoT) [8] effectively will speed up the development and deployment of lightweight JavaScript virtual machines.

The unique characteristics of resource-constrained application domains, such as limited memory, low power, and heterogeneous hardwares, pose distinctive challenges to the implementation of a JavaScript virtual machines [9]. First, the binary size and memory footprint of the JavaScript virtual machines should be small, due to the limited storage space. Second, energy consumption should be low, as battery power is often the primary energy source for these devices. Third, high efficiency is essential to guarantee operation smoothness even on devices with limited computation capability.

Unfortunately, although prior studies [10] [11] [12] [13] have investigated browser-side or sever-side JavaScript virtual machines (*e.g.*, V8 [14], SpiderMonkey [15], and Chakra [16]) extensively, few studies have been conducted on lightweight JavaScript virtual machines deployed in resource-constrained domains. Specifically, to the best of our knowledge, there has not been a comprehensive empirical investigation into lightweight JavaScript virtual machines, in terms of the support for ECMAScript standard, performance, resilience, and code quality. While several studies have been conducted to compare empirically lightweight JavaScript virtual machines, [2] [17], three key issues remain largely unexplored: first, a comprehensive evaluation dataset is still lacking. Previous studies have only proposed partial dataset to measure ECMAScript standard or performance. Without such a comprehensive evaluation dataset, the effectiveness of the testing process can be significantly impacted. Test cases that are poorly designed or do not cover all possible scenarios can lead to false positives or false negatives.

Second, ECMAScript standard support of lightweight JavaScript virtual machines has not been thoroughly investigated. It is critical that lightweight JavaScript virtual machines supports ECMAScript standard, as it determines to what extent JavaScript legacy code can be migrated to these virtual machines. Unfortunately, existing studies [17][18] have demonstrated existing virtual machines (*e.g.*, Quad-wheel[19], Duktape [5]) lack support for the latest JavaScript standard

(*e.g.*, built-in objects or methods such as Date or Math). Worse yet, existing testing tools (*e.g.*, [20] [21]) do not support latest JavaScript standards well.

Third, the resilience of lightweight JavaScript virtual machines has not been studied. It is crucial for lightweight JavaScript virtual machines to be resilient to obfuscated JavaScript code, as obfuscation is a widely used anti-reverse engineering technology to distribute JavaScript source code protecting key intellectual properties [22].

Therefore, in this study, we explore the following research questions that remain unanswered related to the lightweight JavaScript virtual machines. What is the evaluation dataset of lightweight JavaScript virtual machines? To what degree these lightweight JavaScript virtual machines support the ECMAScript standard? What is the performance of lightweight JavaScript virtual machines? How resilient are these virtual machines in executing obfuscated JavaScript programs? What is the code quality of these lightweight JavaScript virtual machines? Without such knowledge, JavaScript developers cannot effectively use these virtual machines for secure and efficient code development, and engine developers may ignore the performance pitfalls that can be optimized.

Our work. In this paper, to fill this gap, we present the *first* and most *comprehensive* empirical study of lightweight JavaScript virtual machines. We first designed and implemented novel software tool prototype JASMIN to conduct this study. 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 virtual machines: QuickJS [3], JerryScript [4], Duktape [5], and MuJS [6]. Finally, we perform an empirical study in terms of ECMAScript standard support, performance, resilience, and code quality.

We obtained important findings and insights from these empirical results, such as: 1) we investigated the ECMAScript standard support of these virtual machines and proposed 3 failure factors; 2) we proposed 3 issues by executing obfuscated JavaScript code in these virtual machines and revealed 3 root causes; and 3) we investigated the code quality of these virtual machines and presented quantitative results.

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 virtual machines to develop effective and compliant JavaScript program; and 2) help virtual machines developer further improve virtual machines, by increasing the ECMAScript standard support, performance, resilience and code quality.

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

- **Empirical study and tools.** We present the first empirical study of lightweight JavaScript virtual machines with a novel software prototype JASMIN we created.
- **Datasets.** We created a comprehensive dataset for evaluating lightweight JavaScript virtual machines.
- **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 presents necessary background knowledge for this work.

A. JavaScript and ECMAScript standard

JavaScript. JavaScript has been one of the most popular programming languages for over two decades [23]. Originally developed by Netscape in 1995 for adding dynamic functionality to Web browsers [24], it is now used for a wide variety of domains, including servers [25], mobile applications, games, and desktop applications. Due to its technical advantages of hot module replacement and event-driven programming, JavaScript is particularly well-suited for resource-constrained application scenarios [17] [26], by leveraging its powerful ecosystem including libraries, frameworks, and toolkits.

ECMAScript standard. The ECMAScript standard [27] is the official specification defining the syntax, semantics, and APIs of the JavaScript programming language. Created by the European Computer Manufacturers Association (ECMA) and first published in 1997 as ECMA-262 [28], the ECMAScript standard is the authoritative guideline for implementing JavaScript virtual machines on diverse underlying platforms, and has been supported by all major JavaScript virtual machines including V8, SpiderMonkey, JavaScriptCore [29]. Specifically, ECMAScript standard has a regular update cycle, with new versions released on a annual basis, putting forward higher requirements for virtual machines implementations.

B. Lightweight JavaScript Virtual Machines

Lightweight JavaScript virtual machines have been specifically engineered to cope with constrained-resource environments such as embedded systems and IoT devices. Specifically, these virtual machines are designed with 3 key design rationals: efficient memory usage, speedy interpreter with minimal start-up time, and high portability through embeddability. Several widely adopted lightweight JavaScript virtual machines are available today, including QuickJS, JerryScript, Duktape, and MuJS. Each of these virtual machines has been developed with specific policies and methodologies designed to adapt to resource-constrained environments. First, these

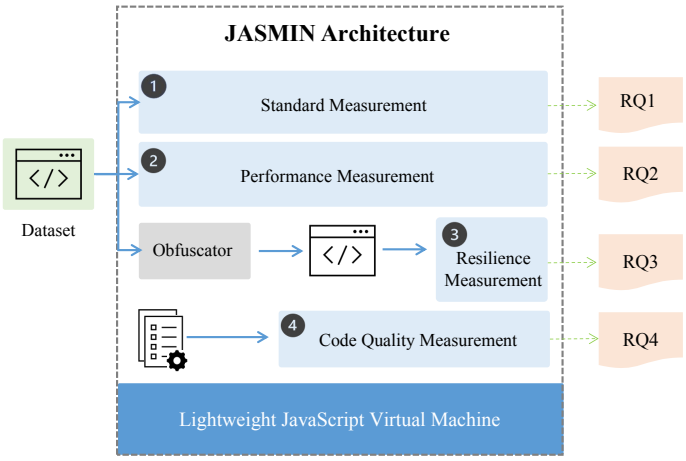

Fig. 1: JASMIN Architecture

virtual machines are designed to minimize memory consumption. They have a small binary file size, allowing them to operate with less RAM. They also use byte code as input data for the interpreter and lack Just-in-Time (JIT) support to reduce memory overhead. Second, these virtual machines are built to be highly portable and therefore require only a few system libraries, enabling them to operate on various platforms without significant customization.

The lightweight JavaScript virtual machines usually consist of 2 main components: parser and virtual machine. The parser undertakes lexical analysis of the input JavaScript source code to generate an intermediary language, typically a byte code that is specifically customized to the engine. Subsequently, the virtual machine executes the sequential byte code instructions.

Many projects have been developed that utilize lightweight JavaScript virtual machines, including game virtual machines, static compilers, IoT frameworks, SQLite, monitors, and cross-platform user interface (UI) development libraries [30] [31] [32]. These projects make use of lightweight JavaScript virtual machines to provide an efficient, fast and low-resource alternative to traditional JavaScript virtual machines, enabling resource-constrained environments to operate effectively and efficiently.

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), and the code quality measurement module (Section III-F), respectively.

A. Design Goals

The design goals of JASMIN is to achieve automaticity and scalability. Specifically, it is challenging to perform an empirical study of lightweight JavaScript virtual machines with a comprehensive evaluation metrics, for two key reasons: 1) automation and 2) scalability. First, the study should be fully automated, otherwise it is difficult, if not impossible, to evaluate comprehensive metrics of lightweight JavaScript virtual machines in a fully automatic manner; human analysis is only required to complement the analysis by manual code inspection. Second, this analysis should be scalable to analyze different lightweight JavaScript virtual machines, even potential ones in the future.

B. The Architecture

Based on this design goals, we present, in Fig. 1, the architecture of JASMIN, consisting of four key modules. First, the standard measurement module (❶) takes as input ECMAScript standard datasets, assesses the support of lightweight JavaScript virtual machines, and generates reports on ECMAScript standard support. It is used to answer the first research question (**RQ1**).

Second, the performance measurement module (❷) measures the binary file size and the start-up time of each virtual machines, then evaluate the execution time, and heap size of lightweight JavaScript virtual machines, to produce a performance report, which is used to answer the second research question (**RQ2**).

Third, the resilience measurement module (❸) takes as input a JavaScript program, obfuscates it with a given obfuscator to produce a obfuscated JavaScript program. Then the two JavaScript programs are executed to compare the outputs with a differential testing approach. The measurement result is used to answer the third research question (**RQ3**).

Finally, the code quality measurement module (❹) uses to measure the code smell and cyclomatic complexity of lightweight JavaScript virtual machines and outputs the reports, which is used to answer the fourth research question (**RQ4**).

In the following sections, we discuss the design and implementation of each module, respectively.

C. Standard Compatibility Measurement

The standard measurement module generates ECMAScript standard support results by running the input ECMAScript standard dataset on lightweight JavaScript virtual machines 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 includes 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).

D. The Performance Measurement Module

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

The performance measurement module employs a performance benchmark dataset to assess the efficiency of the lightweight virtual machines. 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, running 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).

E. The Resilience Measurement Module

We have incorporated the resilience module into the JASMIN to evaluate the resilience of lightweight JavaScript virtual machines for obfuscated JavaScript dataset 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).

F. The Code Quality Measurement Module

Code smells, which are signs of low code quality that may negatively impact maintainability, are commonly used indicators for assessing code quality [33]. The detection and elimination of code smells can also help prevent the occurrence of bugs [34].

The cyclomatic complexity (CC) metric measures the number of linearly independent paths through a piece of code [35]. It is widely cited as a useful predictor of various software attributes such as reliability and development effort [36].

We included the code quality module in JASMIN, by employing the static analysis tool OCLint [37] to determine the number of code smells and the number of cyclomatic complexity (CC) per project.

To address the potential confounding effect of project size, we evaluated code quality for each individual project by calculating the ratio of code smells per line of code (LoC), as well as the ratio of cyclomatic complexity (CC) per line of code (LoC). The Code Quality module has been introduced to address **RQ4** (Section IV-A).

TABLE I: JavaScript virtual machines leveraged by JASMIN.

Name	License	Implementation Language	Open Source
QuickJS [3]	MIT	C	✓
JerryScript [4]	Apache-2.0	C	✓
Duktape [5]	MIT	C	✓
MuJS [6]	ISC	C	✓

IV. EMPIRICAL RESULTS

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

A. Research Questions

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

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

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

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

RQ4: Code quality. What is the code quality of these JavaScript virtual machines?

B. 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.

C. Lightweight JavaScript Virtual Machines and Datasets

We first describe lightweight JavaScript virtual machines selected, and datasets created in this study.

Lightweight JavaScript virtual machines. Following prior studies [11], we selected lightweight virtual machines according to the five criterion that the virtual machine 1) has more than 700 stars on GitHub, representing the popularity, 2) is developed mainly in C (or C++), with 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, obfuscation, and code quality, 3) is still actively updated and maintained after April 1, 2021 (*i.e.*, update during the past two years), 4) is releases more than 3 versions, and 5) is open source code, which facilitate us to investigate the root causes of the research results. As a result, We have selected 4 lightweight JavaScript

TABLE II: Datasets used in our study.

Name	DS1		DS2	DS3	
	DS-ES5	DS-ES14		sub-bef	sub-aft
#No.	11,725	93,751	14	950	950

virtual machines: QuickJS, JerryScript, Duktape, and MuJS, as TABLE I presented, in our study.

Datasets. To conduct the empirical study, we created 3 datasets **DS1**, **DS2** and **DS3**, as shown in TABLE II. First, we created a dataset **DS1** to measure the ECMAScript standard conformance with 105,476 benchmarks, by leveraging the Test262 [38], an official ECMAScript conformance test suite. We then divide this dataset into two sub-datasets: 1) **DS-ES5**, a dataset of the ECMAScript 5 edition with 11,725 benchmarks; and 2) **DS-ES14**, a dataset of the ECMAScript 2023 edition (*i.e.*, 14) with 93,751 benchmarks. This division has been necessitated by the significant dissimilarities between the ECMAScript 5 edition and the ECMAScript 6 edition and its subsequent iterations. Specifically, the ECMAScript 6 edition was the first major update to the ECMAScript, introducing a plethora of novel syntax, operators, primitives, and objects. Commencing with the ECMAScript 6 edition, an annual version release strategy was adopted to enable the language to mature consistently and progressively over time [39].

Second, DS2 is a performance benchmark. We selected a set of benchmark suites that came from SunSpider [40], Octane 2 [41], Kraken [42], and JetStream 2 [43], 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 [44][12]. Furthermore, we have removed test cases that are not suitable for lightweight implementation goals, such as 3D rendering and other computationally intensive tests. A final set of 14 cases that encompass 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 DS2.

Finally, D3 is an obfuscated JavaScript dataset including the unobfuscated JavaScript dataset as sub-bef and the obfuscated JavaScript dataset as sub-aft. We created the D3 with the following steps: 1) To guarantee the diverse of JavaScript tests, it is imperative that the tests should be random, comprehensive, and covering a broad range of ECMAScript language features. To achieve this, we have selected 950 JavaScript tests as sub-bef from the feature test files of these four lightweight JavaScript virtual machines' projects; 2) To generate obfuscated JavaScript tests, we utilized a widely-used and highly effective obfuscation tool, known as UglifyJS [45] that has been used in prior works [46][47][48], 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 sub-aft.

D. RQ1: ECMAScript Standards Support

To answer **RQ1** by investigating the ECMAScript standards conformance of lightweight JavaScript virtual machines, we applied JASMIN to the dataset **DS1** (including both **DS-ES5** and **DS-ES14** in TABLE II). The DS-ES5 dataset is used to evaluate all four virtual machines, whereas DS-ES14 dataset is used to evaluate all four virtual machines except for MuJS, as it only supports for ECMAScript 5

TABLE III presents the empirical results of ECMAScript Standard 5 edition support in the four JavaScript virtual machines. The first column lists all JavaScript virtual machines. The next 2 columns present the numbers of failed benchmarks as well as success rates.

The empirical results give interesting findings and insights. First, except for MuJS (with a success rate of 83.67%), the other 3 JavaScript virtual machines 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 3 key reasons (as presented by the last 3 columns in TABLE III): 1) ECMAScript edition distinctions; 2) Unicode edition discrepancies; and 3) unsupported JavaScript features. First, several JavaScript virtual machines 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 an in ECMAScript 5, leading to infinite execution of the following JavaScript program.

```
1 var objOne = { 0: true, 1: true, length: "
      Infinity" };
2 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 virtual machines. TABLE IV presents the empirical results of ECMAScript Standard 2023 edition support in QuickJS, JerryScript, and Duktape. Test the virtual machines' standard support across four aspects: Annex B, 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 virtual machines have a poor support in intl402 and annexB.

TABLE III: Success rates, failures, failure factors for the 4 JavaScript virtual machines, on the ECMAScript Standard 5/5.1 edition dataset.

JavaScript virtual machines	Result		Failure Factors		
	Failed Numbers	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

TABLE IV: Pass files of the DS-ES14.

	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

TABLE V: JavaScript features supported poorly by QuickJS and JerryScript.

	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 VI: Binary file size (KB).

	QuickJS	Duktape	MuJS	JerryScript
Binary Size	1,126	341	352	440

Third, both QuickJS and JerryScript also exhibit poor support for 9 features including internationalization API, temporal, the waitAsync of atomics, the prototype of atomics, the prototype of shadowrealm, the waitAsync of language, regexp, the top-level-await of language, the top-level-await of built-ins as shown in TABLE V.

Summary: All four virtual machines 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 virtual machines. As a result, developers must exercise caution when implementing these features in their programs to avoid compatibility issues.

E. RQ2: Performance

To answer **RQ2** by investigating the performance of lightweight JavaScript virtual machines, we firstly compared the binary file size of the four JavaScript virtual machines. Then calculated the cold start-up time. Finally, we applied JASMIN to the DS2 dataset (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 other three lightweight JavaScript virtual machines typically do not exceed 500 KB, making it very suitable for 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 virtual machines. The outcomes demonstrate that each of the four virtual machines exhibits a remarkably swift start-up speed, which does not exceed 9 ms.

TABLE VIII presents the execution time results of running the DS2 dataset. First, On average, QuickJS (with the average execution time of 80.4 ms) outperforms the other three virtual machines 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 other three virtual machines by more than 35 times.

TABLE IX presents the heap size results of running the DS2 dataset. First, JerryScript (with the average heap size of 106.3 KB) demonstrates a notable advantage over the other three virtual machines 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 virtual machines by a factor of 177.

We then explored the root causes, based on a manual inspection of the JavaScript virtual machines' sources. This inspection revealed 2 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 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 of the 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.

TABLE VII: Start-up time (ms).

	QuickJS	Duktape	MuJS	JerryScript
Start-up time	2	9	3	2

TABLE VIII: Execution time (ms).

	QuickJS	Duktape	MuJS	JerryScript
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

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.

F. RQ3: Resilience

To answer **RQ3** by evaluating the resilience of those lightweight JavaScript virtual machines, we applied JASMIN to the **DS3** dataset, *i.e.*, the obfuscated JavaScript programs.

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 sub-aft) demonstrated the highest resilience among the four virtual machines compared, as the execution results remain unchanged before and after obfuscation. Second, QuickJS and Duktape (with 1 new failure number of running sub-aft) exhibited remarkable resilience by following JerryScript. Third, it seems like MuJS has experienced 3 failures when executing the sub-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 virtual machines, hence causing failures in executing the obfuscated JavaScript code. For example, Duktape does not support the feature of setting prototype

TABLE IX: Heap size (KB).

	QuickJS	Duktape	MuJS	JerryScript
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

with `__proto__` in an object expression, which is present in ECMAScript 6. This is exemplified in the following code:

```
1 var n=[13.37],
2 n={a:n,length:n,
3 __proto__:[13.37,13.37,13.37]};
4 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 regulations of MuJS on the limit of AST nesting:

```
1 // jsi.h: line 104
2 #define JS_ASTLIMIT 100  /* max nested
      expressions */
3 // jsparse.c: line 24
4 #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:

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

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

Summary: All four lightweight virtual machines, 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.

G. RQ4: Code Quality

To answer **RQ4** by evaluating the code quality of those lightweight JavaScript virtual machines. we applied JASMIN to analyze the source code of these engine projects for code smells and cyclomatic complexity (CC).

TABLE XI presents the cyclomatic complexity (CC), code smells and violation priority for the four virtual machines. The decreased code complexity and minimized code smells are indicators of higher code quality.

The empirical results give interesting findings and insights. First, except for MuJS, JerryScript exhibits the lowest cyclomatic complexity and code smell, followed by QuickJS and Duktape. Second, we speculate that MuJS's superior performance in terms of cyclomatic complexity and code smell is associated with its smaller overall code volume. The relatively reduced code size of MuJS intuitively results in lower cyclomatic complexity and code smell compared to the other three virtual machines.

Summary: QuickJS, JerryScript, and Duktape share a comparable code volume, and among them, JerryScript demonstrates superior code quality. Meanwhile, MuJS also exhibits commendable code quality, which could potentially be attributed to the project's reduced size.

V. IMPLICATIONS

This work presents the first and most comprehensive empirical studies of lightweight JavaScript virtual machines that have actionable implications for several audiences. This section discusses some implications of this work along with some important directions for future studies.

For JavaScript developers. Compared with traditional JavaScript virtual machines, lightweight JavaScript virtual machines emerged at a later stage and are still undergoing rapid development. The study provides valuable insights into the standard support, performance, flexibility, and code quality of lightweight JavaScript virtual machines. By gaining a deeper understanding of these key aspects, JavaScript developers are empowered to make informed decisions when selecting an appropriate JavaScript engine and to develop optimized and compatible code tailored to the specific features of the chosen engine.

Furthermore, the prototype system JASMIN we provided offers JavaScript developers the opportunity to test the latest JavaScript engine's standard support. This feature can allow them to apply the most new JavaScript features to their projects in a timely manner.

The findings of this research hold 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 virtual machines face a challenging task of balancing the performance and memory consumption of their virtual machines to cater to resource-constrained scenarios. In such situations, it is crucial for developers to optimize their virtual machines to ensure efficient resource utilization while still maintaining high performance standards.

The results of this research can provide valuable insights to developers of lightweight JavaScript virtual machines regarding the optimization of their virtual machines' performance and memory consumption. By analyzing the data obtained from our prototype system, developers can identify the areas in their code that require optimization and subsequently fine-tune their virtual machines accordingly. In addition, this research can also help developers to identify and fix code errors that may impact the resilience and code quality of their virtual machines.

By following the optimization directions and fixing the code errors highlighted in this research, developers can make their virtual machines are well-suited to a wide range of use cases.

For engine researchers. The empirical study can help engine researchers in multiple ways. First, the findings can contribute to the development of standardized lightweight engine byte code that effectively balances memory usage and performance optimization. Second, the results may facilitate the formulation of interaction standards between lightweight JavaScript virtual machines and emerging technologies, such as WebAssembly. This can ultimately lead to more efficient and seamless integration of various technologies, thereby enhancing the overall functionality and performance of virtual machines.

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

TABLE XI: The Cyclomatic Complexity (CC), code smells and violation priority for the 4 virtual machines.

JavaScript virtual machines	LoC	CC	CC per LoC	Code Smells	Code Smells per LoC	Violation Priority		
						P1	P2	P3
QuickJS	73096	356	0.49%	11650	15.94%	0	2826	9180
JerryScript	99665	411	0.41%	13091	13.13%	0	7024	6478
Duktape	81893	445	0.54%	14436	17.63%	0	7638	7243
MuJS	16673	8	0.05%	136	0.81%	0	30	114

Datasets. In this study, we used three different datasets, namely DS1, DS2, and DS3. While DS3 was created using the tests included in those lightweight JavaScript virtual machines' project, 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.

Other Factors. In this work, we focus on the research of lightweight virtual machines in terms of features, performance, resilience and code quality. Given that lightweight virtual machines are frequently employed in situations where resources are limited, they typically engage in some degree of input code optimization, which can compromise code legibility and debuggability. Moving forward, improving the readability and debuggability of optimized code is an interesting direction for future investigation.

VII. Related Work

There is a significant amount of research effort on lightweight JavaScript virtual machines. However, the work in this paper represents a novel contribution to this research field.

Lightweight JavaScript Virtual Machines Studies. Lightweight JavaScript virtual machines has been studied extensively. Sin D et al. [2] proposed several IoT workloads to evaluate the performance and memory overhead of IoT systems, and evaluate several lightweight JavaScript frameworks. In addition, they evaluated the effectiveness of multicore system for JavaScript framework. Kim M et al. [17] analyzed the performance of three lightweight JavaScript virtual machines including Quad-wheel, Espruino, and Duktape. They also proposed several optimisation ideas for Duktape and implemented the suggested ideas. Finally, their study has achieved an increase of performance and a reduction of memory footprint. Park H et al. [49] unearthed that a substantial portion of the heap memory is allocated for JavaScript source code, particularly in lightweight JavaScript virtual machines. To resolve this memory issue, they proposed an optimization called dynamic code compression to reduce the memory consumption of the source code in JavaScript. Their work has been successfully merged into the main branch of the Escargot [50] engine. However, they do not conduct large-scale empirical studies on lightweight JavaScript virtual machines.

The Empirical Study of JavaScript Virtual Machines. Ziyuan Wang et al. [11] performed the first empirical study on bugs in three mainstream JavaScript virtual machines, including V8, SpiderMonkey, and Chakra, to collect and classify their bugs. They found that the compiler and the DOM are the most buggy component in V8 and SpiderMonkey, respectively. And the semantic bugs are the most common root causes of bugs. Park S et al. [13] designed and implemented CRScope, which automatically classified security and non-security bugs from given crash-dump files to determine whether an observed crash triggers a security bug in JavaScript virtual machines. The extant empirical studies pertaining to JavaScript virtual machines predominantly concentrate on mainstream and large-scale virtual machines, with a focus on the research and analysis of bugs. However, the ECMAScript standard support, performance, resilience and code quality of lightweight JavaScript virtual machines have not been thoroughly assessed.

VIII. Conclusion

In this work, we presented the first and most comprehensive study of lightweight JavaScript virtual machines. By designing and implementing a software prototype JASMIN, we evaluated four lightweight JavaScript virtual machines in terms of ECMAScript standard support, performance, resilience and code quality. We found the underlying reasons behind the lack of support for ECMAScript standard and investigated the factors that affect performance. Additionally, we explored and analyzed the root causes for the resilience of executing obfuscated JavaScript code, as well as evaluated the code quality of lightweight JavaScript virtual machines. Our recommendations are directed towards JavaScript developers and engine developers, and their implementation can help to create a more robust ecosystem for lightweight JavaScript virtual machines.

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. Cham: Springer International Publishing, 2016, pp. 602–609.

[3] "Quickjs javascript engine," https://bellard.org/quickjs/.

[4] "Jerryscript," https://jerryscript.net/.

[5] "Duktape," https://duktape.org/.
[6] "Mujs," https://mujs.com/.
[7] bellard, "Bellard/quickjs," Apr. 2023.
[8] L. Sciullo, L. Gigli, F. Montori, A. Trotta, and M. D. Felice, "A survey on the web of things," *IEEE Access*, vol. 10, pp. 47 570–47 596, 2022.
[9] A. Taivalsaari and T. Mikkonen, "On the development of iot systems," in *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. Barcelona: IEEE, Apr. 2018, pp. 13–19.
[10] 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)*. Florianopolis, Brazil: IEEE, Oct. 2015, pp. 41–48.
[11] Z. Wang, D. Bu, N. Wang, S. Yu, S. Gou, and A. Sun, "An empirical study on bugs in javascript engines," *Information and Software Technology*, vol. 155, p. 107105, Mar. 2023.
[12] H.-K. Choi and J. Lee, "Optimizing constant value generation in just-in-time compiler for 64-bit javascript engine," *Journal of KIISE*, vol. 43, no. 1, pp. 34–39, 2016.
[13] 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. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 646–657.
[14] "V8 javascript engine," https://v8.dev/.
[15] "Spidermonkey," https://spidermonkey.dev/.
[16] "Chakracore," chakra-core, Apr. 2023.
[17] G. Keramidas, N. Voros, and M. Hübner, Eds., *Components and Services for IoT Platforms*. Cham: Springer International Publishing, 2017.
[18] 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)*. Madrid, ES: IEEE, May 2021, pp. 13–24.
[19] "Google code archive - long-term storage for google code project hosting." https://code.google.com/archive/p/quad-wheel/.
[20] "Test262 report," https://test262.report/.
[21] "Test262-harness," https://www.npmjs.com/package/test262-harness, Jun. 2022.
[22] 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*. Fajardo, PR, USA: IEEE, Oct. 2012, pp. 9–16.
[23] "Tiobe index," https://www.tiobe.com/tiobe-index/.
[24] A. Wirfs-Brock and B. Eich, "Javascript: The first 20 years," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–189, Jun. 2020.
[25] 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*. Salamanca Spain: ACM, Apr. 2015, pp. 798–801.
[26] "Javascript empowered internet of things — ieee conference publication — ieee xplore," https://ieeexplore.ieee.org/abstract/document/7724687.
[27] E. C. M. Association (ECMA), "Ecmascript language specication," *http://www.ecma-international.org/publications/standards/Ecma-262.htm*.
[28] C. Dittamo, V. Gervasi, E. Boerger, and A. Cisternino, *A Formal Specification of the Semantics of ECMAScript*. Pisa, IT: Università di Pisa, Feb. 2011.
[29] "Javascriptcore — apple developer documentation," https://developer.apple.com/documentation/javascriptcore.
[30] "Iot.js," https://iotjs.net/.
[31] "Introducing javascript* runtime for zephyr™ os - zephyr project," https://www.zephyrproject.org/introducing-javascript-runtime-for-zephyr-os/.
[32] https://wiki.duktape.org/projectsusingduktape.
[33] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. Koblenz, Germany: IEEE, Oct. 2013, pp. 242–251.
[34] ——, "Do code smells reflect important maintainability aspects?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento, Italy: IEEE, Sep. 2012, pp. 306–315.
[35] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE Software*, vol. 33, no. 6, pp. 27–29, Nov. 2016.
[36] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, p. 30, 1988.
[37] "Oclint," https://oclint.org/.
[38] "Tc39/test262," Ecma TC39, Apr. 2023.
[39] J. D. Isaacks, *Get Programming with JavaScript Next: New Features of ECMAScript 2015, 2016, and Beyond*. Simon and Schuster, Apr. 2018.
[40] "Sunspider 1.0 javascript benchmark," http://proofcafe.org/jsx-bench/js/sunspider.html.
[41] "Octane 2.0 javascript benchmark," https://chromium.github.io/octane/.
[42] "Kraken - mozillawiki," https://wiki.mozilla.org/Kraken.
[43] "Jetstream 2," https://browserbench.org/JetStream2.0/.
[44] J. Radhakrishnan, "Hardware dependency and performance of javascript engines used in popular browsers," in *2015 International Conference on Control Communication & Computing India (ICCC)*, Nov. 2015, pp. 681–684.
[45] "Uglifyjs — javascript parser, compressor, minifier written in js," https://lisperator.net/uglifyjs/.
[46] G. S. Ponomarenko and P. G. Klyucharev, "On improvements of robustness of obfuscated javascript code detection," *Journal of Computer Virology and Hacking*

Techniques, Sep. 2022.

[47] S. Rauti and V. Leppanen, "A comparison of online javascript obfuscators," in *2018 International Conference on Software Security and Assurance (ICSSA)*. Seoul, Korea (South): IEEE, Jul. 2018, pp. 7–12.

[48] 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*, May 2014, pp. 489–496.

[49] 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.

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