A Survey of Fuzzing Tools for JavaScript Engines

Bruno Gonçalves Oliveira[1], **Silvia Regina Vergilio**[1], **André T. Endo**[2]

[1] Federal University of Paraná (UFPR)
CP 15.064 – 91.501-970 – Curitiba – PR – Brazil

[2]Federal University of São Carlos (UFSCar)
São Carlos – SP – Brazil

brgoncalves@gmail.com, silvia@inf.ufpr.br, andreendo@ufscar.br

Abstract. *JavaScript is a critical programming language to Web applications. Code written in this language is processed and executed by specialized engines. Because these engines are complex and may contain vulnerabilities that enable attacks on end-user browsers or server-side applications, fuzzing has become the main technique to uncover security issues. Nevertheless, there is a lack of studies that characterize existing fuzzing tools for JavaScript engines and their main particularities. In this paper, we survey state-of-the-art fuzzing tools for JavaScript engines. For the 19 fuzzers identified, we analyze several dimensions related to the fuzzing steps, implementation details, and the benchmarks used. A variety of fuzzing techniques are employed, making the fuzzers more efficient and targeting specific classes of vulnerabilities. Popular JavaScript engines such as ChakraCore, JavaScriptCore, V8, and SpiderMonkey have been extensively tested using existing fuzzers. Finally, we discuss research insights to support researchers and practitioners in the development of new fuzzers for JavaScript engines.*

1. Introduction

JavaScript engines are essential components of modern web browsers and many other applications. They are responsible for parsing, interpreting, compiling, and executing JavaScript code on dynamic web pages. These engines perform on-the-fly compilation rather than pre-compiling code. Two notable characteristics of JavaScript are its support for multiple programming paradigms and dynamic typing [Mikkonen and Taivalsaari, 2007], where variable types are not resolved at compile time but during execution [Kang, 2021]. These characteristics enhance performance but, on the other hand, are associated to unique or distinct security vulnerabilities. Such vulnerabilities arise from real-time compilation, the use of memory-unsafe programming languages in engine development, and runtime-specific features that can lead to nondeterministic behavior [Wang, 2017]. Due to JavaScript's runtime variability, repeated executions of the same code may produce different outcomes, introducing unique challenges and risks. Common vulnerabilities found in JavaScript engines include memory mismanagement, improper input validation, and unsafe type conversions [Groß et al., 2023; Kang, 2021; Lee et al., 2020]. These security issues may be exploited by malicious code that can arbitrarily trigger unexpected behavior, including remote command execution in the browser's user machine.

Given the critical role JavaScript engines play in modern applications, securing them is essential to protect users and systems. Software testing is one method for ensuring this protection. Complementing these efforts, researchers have explored a variety of

strategies, including static analysis, secure development practices[1], and code review tools such as Snyk[2]. Among these efforts, fuzzing has emerged as one of the most effective techniques for identifying security vulnerabilities in complex software systems [Yu et al., 2024]. Fuzzing is a dynamic testing method that automatically generates and executes a huge amount of inputs to uncover unexpected or unsafe program behavior [Miller et al., 1990]. It systematically probes software components to trigger scenarios that might lead to crashes, memory corruption, or logic errors. However, the fuzzing process requires automatic support.

Fuzzers, or fuzzing tools, automatic generate inputs, usually with seed generators, which produce data or code fragments designed to interact with a target program and expose its behavior. These tools initiate fuzzing campaigns, which are automated testing sessions in which generated inputs are continuously fed to the target program. During the campaign, the fuzzer monitors how the program responds (e.g., via coverage metrics or crash signals) to guide further input generation. As we can see, fuzzers are fundamental tools to the practical use of fuzzing in real and industrial scenarios. Moreover, due to the particularities and importance of JavaScript, the interest in fuzzers for Javascript engines is crescent.

Motivated by these facts, in this work, we survey state-of-the-art fuzzing tools for JavaScript engines, analyze the implemented fuzzing techniques and input generation strategies, vulnerability detection methods, and compare their technical characteristics. For the 19 fuzzers identified, we observe that most of them target general vulnerabilities, use *Proof-of-Concept (PoCs)* as seeds, adopt hybrid strategies for input generation, employ code coverage to guide the fuzzing process, and use crashes as oracle. To evaluate the fuzzers, 10 different JavaScript engines are taken into account; ChakraCore, JavaScriptCore, V8, SpiderMonkey and JerryScript are the most frequently used in the studies. We also discuss the main research gaps and limitations of the tools, as well as trends in the area. In this way, the main contributions are twofold: *(i)* to help researchers in the identification of research opportunities and encourage new works on this subject to solve existing challenges and offer to the software industry cost-effective fuzzing tools; and *(ii)* to provide knowledge for both, beginners and advanced developers, to better understand fuzzing.

The remaining of this paper is organized as follows. Section 2 overviews background on JavaScript engines and fuzzing testing, as well as related work. Section 3 describes how the research was conducted. Section 4 presents the main outcomes of our survey. Section 5 summarizes the main findings of our study, discussing open research directions. Section 6 concludes the paper.

2. Background and Related Work

2.1. JavaScript engines

In modern web applications, JavaScript engines are fundamental components that allow parsing, interpreting, compiling, and executing JavaScript code. They help to provide not only rich pages in the user front-end by powering web browsers, but also server-side

[1] `https://www.ncsc.gov.uk/collection/developers-collection`
[2] `https://www.snyk.io/`

applications built on top of runtimes like Node.js and Deno. The JavaScript engines interpret and compile code at runtime, typically implementing an architecture that comprises a parser, an interpreter, a baseline compiler, and a *Just-In-Time (JIT)* compiler or optimizer. The JavaScript compilation process begins when the parser generates an *Abstract Syntax Tree (AST)* from the source code [Disney et al., 2014]. This AST is then handed to the interpreter that prepares the code for compilation. The compiler takes the result from the interpreter and converts it into bytecode. Finally, the JIT compiler can optimize this bytecode using runtime profiling data to improve execution performance [Kienle, 2010].

Because JavaScript engines are often implemented in unsafe low-level programming languages like C and C++, they are prone to memory corruption issues. While these languages provide essential performance benefits for real-time compilation, they also introduce exploitable issues, and leads to several classes of vulnerabilities [Rodrigues et al., 2019]. Common memory issues include dangling pointers, which reference freed memory and may cause crashes or unauthorized access; heap metadata corruption, which disrupts memory management routines; and uninitialized reads, which can expose unpredictable or sensitive data. Buffer overflows and out-of-bounds accesses can overwrite adjacent memory, leading to control flow hijacking. Other critical flaws include use-after-free (UAF), where deallocated memory is erroneously reused; double frees, which destabilize memory allocators; and type confusion, where incorrect type assumptions, often during JIT optimizations, allow access to objects with incompatible representations [Kang, 2021; Rodrigues et al., 2019].

JavaScript engines exhibit additional complexity due to inconsistent interpretations across implementations. A program may behave differently on different engines, complicating cross-platform fuzzing efforts. Techniques such as differential testing and test transplantation have been used to identify these inconsistencies [Lima et al., 2021]. To summarize, JavaScript engine vulnerabilities arise from four primary factors: open-source exposure, development in unsafe languages, dynamic typing, and diverging engine architectures. Static analysis is often impractical due to the size and complexity of engine codebases. Fuzzing, as a dynamic analysis technique, remains one of the most effective ways to uncover bugs, even when researchers lack detailed knowledge of internal mechanisms.

2.2. Fuzzing

Fuzzing is the process of submitting random inputs to a program to observe unexpected behaviors. Its first evaluation, for Unix utilities, revealed numerous issues triggered by random, oversized input sequences [Miller et al., 1990]. Since then, fuzzing has evolved significantly, adopting various approaches tailored to the complexity of different software domains. Despite this evolution, the core idea of stress-testing programs through input variation remains central.

Fuzzing has proven to be an effective and scalable technique for discovering vulnerabilities, including in JavaScript engines, and is a technique widely adopted by major vendors such as Mozilla, Google, and Microsoft. The primary challenge lies in generating inputs (i.e., JavaScript programs) that are both syntactically correct and semantically valid. Invalid inputs are typically rejected during parsing or analysis, never reaching the execution phase where most bugs reside [Sun et al., 2022].

JavaScript code execution typically progresses through three stages: parsing, semantic analysis, and runtime execution. Because inputs are often rejected during the first two stages, fuzzers must generate well-formed inputs that reach runtime to uncover deeper bugs [Bernhard et al., 2022]. Simple random mutations rarely achieve this. Effective fuzzing workflows address this challenge through several key components: knowledge of the target software, input generation, input preprocessing, error detection, code coverage tracking, and input evaluation [Park et al., 2020]. These components are illustrated in Figure 1 and explained next.

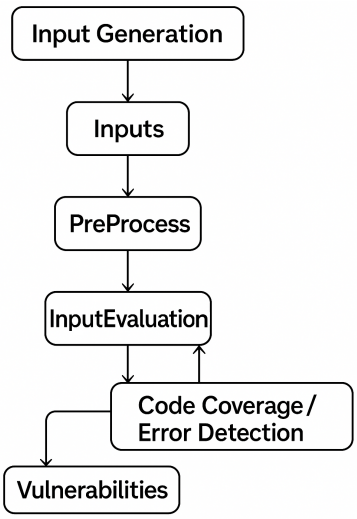

Figure 1. Basic fuzz testing workflow.

The fuzzing process begins with preprocessing, where seed inputs[3] are filtered, trimmed, or normalized to reduce redundancy. Scheduling determines which configuration or input will be tested next. In advanced fuzzers, this step is guided by heuristics based on code coverage, execution time, or previously observed behaviors. Input generation is the foundation of fuzzing. Tools automatically produce a large volume of JavaScript programs with varying structures and complexity to stress different code paths [Park et al., 2020]. Input evaluation, also referred to as the oracle, determines whether a test input reveals a potential issue. Evaluation outcomes may influence configuration updates, adjusting the fuzzer's behavior to enhance efficiency or reach deeper code paths. Seeds form the base of the fuzzing campaign. They may be small code snippets, expressions, or full programs that exercise specific engine behaviors. Fuzzers mutate or expand these seeds to explore new execution paths.

Preprocessing includes converting seeds into representations that allow safe manipulation. A common strategy is converting JavaScript code into an AST [Lin et al., 2019], where each construct becomes a node. ASTs allow controlled replacements and restructuring without breaking syntax [Ndichu et al., 2019]. Most fuzzers manipulate ASTs to preserve structural correctness during mutations [Mozilla, 2019; Wang et al., 2017, 2019]. Fuzzilli [Groß et al., 2023], however, introduced an alternative approach using an *Intermediate Language (IL)* to abstract JavaScript syntax, enabling mutation at

[3]Seed inputs are valid programs elaborated by researchers or collected from benchmark suites and open source projects.

the IL level rather than at AST level.

There are two strategies for generating inputs when fuzzing JavaScript engines: mutational and generational [Lee et al., 2020]. These can also be combined into a hybrid approach. Mutation-based fuzzing starts from existing seed inputs and applies transformations to create new test cases. These transformations introduce small changes while aiming to preserve the input's syntactic validity. This strategy does not require a full understanding of the input format, making it easier to implement, but more difficult to maintain syntactic and semantic correctness. Common mutation strategies include bit flipping, insertion or deletion of tokens, and dictionary-based replacement. Generational fuzzing relies on a specification or grammar to generate inputs from scratch. The goal is to produce inputs that are both syntactically valid and semantically meaningful. These grammars must align with the language specification and generate sufficiently complex inputs to reach deeper logic. Generational fuzzers typically achieve higher coverage for complex formats, although modern mutation-based fuzzers have closed this gap [Wang et al., 2017]. Hybrid fuzzers combine mutation and generation strategies. This allows for flexible exploration: structural diversity from grammars and fine-grained variation from mutation.

Code coverage tracks the extent to which the generated inputs exercise different parts of the target program. It is commonly measured in terms of executed functions, branches, or lines of code. Coverage feedback is often used to guide input generation and prioritize inputs that explore new paths [Groß et al., 2023; Sun et al., 2022]. Tools that measure coverage typically produce reports that highlight unvisited code regions, allowing the fuzzer to focus on under-tested areas. Error detection refers to identifying inputs that cause abnormal behavior in the engine. Errors may arise from invalid inputs, malformed syntax, or inputs that trigger internal faults. Fuzzers must filter out invalid inputs to avoid wasting computational resources [Tian et al., 2021].

The oracle is responsible for the input evaluation and for determining whether an input reveals a potential security issue. This step is crucial for detecting bugs that do not necessarily crash the process but still represent vulnerabilities [Bernhard et al., 2022; Wang et al., 2023]. JavaScript engine fuzzers detect vulnerabilities through various mechanisms. Segmentation faults arise from invalid memory operations, such as null pointer dereferences or division by zero, and are typically captured by system-level signals [Lee et al., 2020]. Beyond crashes, memory safety sanitizers like AddressSanitizer (ASAN) and MemorySanitizer (MSAN) offer deeper analysis by instrumenting the compiled code to uncover issues such as buffer overflows, use-after-free bugs, or uninitialized memory reads, even in cases where the program continues executing without crashing [Stepanov and Serebryany, 2015].

2.3. Related work

In the literature, we can find some pieces of work with goals that are similar to ours. However, the focus of most of them is fuzzing techniques and not tools. The great majority explore fuzz testing in general [Chen et al., 2018; Li et al., 2018; Mallissery and Wu, 2023; Zhao et al., 2024] and not in a specific context. For instance, the survey of Zhu et al. [2022] reviewed the knowledge gaps of general fuzzing, and the main limitations of this technique for vulnerability detection. Differently, our focus is vulnerability identification. Some studies address specific contexts, such as the test of web applications through

web APIs [Dharmaadi et al., 2025; Zhang and Arcuri, 2023]. The work of Daniele et al. [2024] addresses fuzzers for stateful systems and presents a taxonomy for classification and comparison. We have not found works addressing the specific context of JavaScript engines, which is the goal of our work. In this sense, our work focuses deeply on JavaScript particularities, which are not addressed by related work.

3. Tool Selection

Our work has as focus state-of-the-art tools for fuzzing of JavaScript engines, in order to identify possible limitations, research gaps and trends. According to this goal, our review was guided by the following general research question: "What are the main characteristics of existing Javascript engine fuzzers?". The characteristics of interests are those that allow an investigation of how well the tool supports the main components of the fuzzing workflow, presented in the last section (see Figure 1). Based on these characteristics, we established some dimensions to be used in our analysis, as described in Table 1.

Table 1. Dimensions used to analyse the tools

Dimension	Description
input preprocessing	representation strategy
input generation strategy	mutational, generational or hybrid
error detection	if inputs that cause abnormal behavior are identified
code coverage tracking	if code coverage is used to guide the process
input evaluation	type of oracle to determine if the input reveals a potential security issue
used seeds	what is used as source for input generation
target vulnerability	if the process targets specific vulnerabilities
multithread	if supports multithread processing

To select the tools included in this study, we followed a multi-step process. First, we leveraged the first author' expertise and prior experience with security research and JavaScript engine fuzzing to identify well-known approaches. Second, we considered tools repeatedly referenced by multiple authors as state-of-the-art in the literature. Third, we applied a snowballing strategy, inspecting references and related work sections of relevant papers to identify additional tools. Finally, we complemented this process with targeted searches in major academic databases and by reviewing proceedings of conferences in the area to ensure representative coverage. Although snowballing is a recognized method adopted in systematic reviews, the process adopted is a threat to the validity of our results, because it includes some subjective choices in the references and conference proceeding selected.

In the end, we selected 19 tools, whose characteristics are described in Table 2. As a note, all fuzzers selected for this study are white-box tools that integrate with the JavaScript engine source code to interact in real time with execution paths and control flow. They are analyzed in the next section.

4. Tool Analysis

4.1. General overview

The earliest efforts in JavaScript engine fuzzing trace back to 2007 with the release of jsfunfuzz [Mozilla, 2019; Ruderman, 2007] by Mozilla. Designed specifically for the

Table 2. JavaScript Engine Fuzzing Tools Comparison

Tool	Multi-Thread	Target Vulnerability	Seeds	Pre-process	Input Generation	Error Detection	Code Coverage	Input Evaluation
jsfunfuzz [Ruderman, 2007]	No	No	Test Suite	AST	Gen.	No	No	Crashes
LangFuzz [Holler et al., 2012]	No	No	PoCs	AST	Hybrid	No	No	Crashes
IFuzzer [Veggalam et al., 2016]	No	No	Test Suite	AST	Hybrid	No	Yes	Crashes
Skyfire [Wang et al., 2017]	No	No	Samples	AST	Hybrid	No	No	Crashes
CodeAlchemist [Han et al., 2019]	No	No	PoCs, Test Suite	AST	Gen.	No	No	Crashes
DIE [Park et al., 2020]	Yes	Yes	PoCs	AST	Hybrid	Yes	Yes	Crashes
Fuzzilli [Groß et al., 2023]	Yes	No	Initial Corpus	IL	Hybrid	Yes	Yes	Src Instrumented
Deity [Lin et al., 2019]	No	No	PoCs	AST	Hybrid	No	No	Crashes
Superion [Wang et al., 2019]	No	No	Test Suites	AST	Mut.	No	Yes	Crashes
Montage [Lee et al., 2020]	No	No	PoCs, Test Suite	AST	Hybrid	No	No	Crashes
KOP-Fuzzer [Sun et al., 2022]	No	Yes	PoCs	AST	Hybrid	No	Yes	Crashes
JIT-Picker [Bernhard et al., 2022]	Yes	Yes	Initial Corpus	AST	Hybrid	Yes	Yes	Src Instrumented
FuzzJIT [Wang et al., 2023]	Yes	Yes	PoCs	AST	Hybrid	Yes	Yes	Temp
CovRL-Fuzz [Eom et al., 2024]	No	No	PoCs, JS Suite	AST	Mut.	No	Yes	Src Instrumented
PMFuzz [Xu et al., 2024b]	No	Yes	Initial Corpus	AST	Hybrid	No	Yes	Crashes
FuzzFlow [Xu et al., 2024a]	No	No	PoCs	Graph-Based	Hybrid	Yes	Yes	Crashes
Dumpling [Wachter et al., 2025]	No	Yes	PoCs	AST	Hybrid	No	Yes	Src Instrumented
PatchFuzz [Wang et al., 2025]	No	No	PoCs	AST	Mut.	No	Yes	Crashes
JSTargetFuzzer-v1 [Oliveira et al., 2025]	Yes	No	Initial Corpus	IL	Hybrid	Yes	Yes	Src Instrumented

SpiderMonkey engine, it introduced a generational approach to input creation, producing JavaScript constructs such as loops, conditionals, and functions. Despite its simplicity, it was highly effective, uncovering over 1,000 bugs [Holler et al., 2012]. Shortly after,

LangFuzz [Holler et al., 2012] introduced a mutational approach that repurposed code fragments from known PoCs, minimal examples demonstrating how a vulnerability can be triggered, generating inputs through AST expansions guided by the framework ANTLR[4]. IFuzzer [Veggalam et al., 2016] applies genetic algorithms specifically to SpiderMonkey, offering mutator customization and runtime monitoring.

As the field matured, newer fuzzers began to address a common limitation: the prevalence of semantically invalid inputs. In 2017, Skyfire [Wang et al., 2017] extended generational fuzzing with grammar-awareness, using a *Probabilistic Context-Sensitive Grammar (PCSG)* and *American Fuzzy Lop (AFL)* [Google, 2023] to explore syntactically rich inputs. CodeAlchemist [Han et al., 2019] focused on creating valid, both syntactic and semantic correct inputs, noting that nearly 99% of inputs from tools like jsfunfuzz and LangFuzz contained semantic flaws. It transformed ASTs into semantically ordered code blocks to ensure valid recombinations. Similarly, DIE [Park et al., 2020] preserved original data types and structure during mutation, while Fuzzilli [Groß et al., 2023] abstracted JavaScript into an IL and used code coverage to guide evolution via genetic algorithms. Other examples, such as Deity [Lin et al., 2019], used AST-based mutations combined with line-level coverage tracking, and Superion [Wang et al., 2019] added grammar-aware trimming and dictionary-based AST subtree mutations. Neural-guided efforts also emerged, like Montage [Lee et al., 2020], which used language models trained on AST node sequences to generate syntactically consistent subtrees.

A subset of fuzzers concentrated on vulnerabilities tied to JIT compilation. KOP-Fuzzer [Sun et al., 2022] focused on Type Confusion bugs, crafting seed templates from critical engine operations. JIT-Picker [Bernhard et al., 2022] leveraged differential fuzzing to compare execution results with and without JIT optimizations enabled. Likewise, FuzzJIT [Wang et al., 2023] used input templates and a custom oracle to specifically trigger and test optimizer behavior, reporting vulnerabilities in JIT compilers for engines V8, JavaScriptCore, SpiderMonkey, and ChakraCore.

More recent approaches further refine semantic validity and structural precision. PMFuzz [Xu et al., 2024b] integrates a syntax-aware neural program model (SNPM) to create inputs that conform to JavaScript grammar and semantics. FuzzFlow [Xu et al., 2024a] advances differential fuzzing through FlowIR, a control and data flow intermediate representation, enabling semantics-preserving mutations and dynamic dependency tracking. Dumpling [Wachter et al., 2025] focuses on capturing semantic inconsistencies between optimized and unoptimized executions by collecting high-frequency execution state dumps from the V8 engine and identifying divergences with a differential oracle.

Finally, newer fuzzer approaches incorporate large-scale learning techniques and targeted instrumentation. CoVRL-Fuzz [Eom et al., 2024] combines LLMs with coverage-guided reinforcement learning, adapting token mutations based on coverage rewards influenced by TF-IDF scoring. PatchFuzz [Wang et al., 2025], in contrast, emphasizes precision by fuzzing only the code regions affected by recent security-related patches, using PoC extraction and focused instrumentation to reduce overhead and increase the likelihood of discovering regression vulnerabilities. JSTargetFuzzer [Oliveira et al., 2025] uses historical information to guide the fuzzing process to regions more prone

[4]ANother Tool for Language Recognition `https://www.antlr.org`

to contain vulnerabilities.

4.2. Dimension-based analysis

Multithread. As the fuzzing process consumes a lot of resources and takes long time to execute, this is an important characteristic to improve performance. However, we observe that only 5 out of 19 (26%) of the fuzzers supports multi-thread processing. They are Fuzzilli, DIE, JIT-Picker, FuzzJIT and JSTargetFuzzer. The remaining tools operate using a single-threaded model during fuzzing iterations.

Target Vulnerability. The great majority of these tools (13 out of 19 – 68%) adopts a general-purpose approach, aiming to uncover any type of software bug without targeting specific vulnerabilities or behaviors within JavaScript engines. KOP-Fuzzer, however, focuses explicitly on Type Confusion vulnerabilities—a prominent class of issues in JavaScript engines [Sun et al., 2022]. Similarly, JIT-Picker is designed to detect inconsistencies caused by incorrect JIT optimizations using a differential testing strategy, and FuzzJIT also concentrates on the behavior of the JIT compiler and its optimization process. DIE generates inputs based on a strategic template to trigger type confusion vulnerabilities. PMFuzz also targets the optimization process, focusing on JIT (Just-In-Time) compilers that utilize neural networks and JavaScript grammar to generate valid test cases to uncover vulnerabilities. Dumpling also focuses on type confusion issues by comparing non-optimized with optimized runs to identify any inconsistent behavior in the JavaScript engine.

Used Seed. Regarding seed usage, PoCs are the most common input source, employed by 11 (58%) fuzzers. Other tools utilize predefined test cases, language samples, or an initial corpus of JavaScript code as mutation bases.

Input Preprocessing. Sixteen tools (84%) preprocess seeds by parsing them into ASTs, allowing structural transformations during mutation. Fuzzilli and JSTargetFuzzer differ by implementing an IL to abstract JavaScript syntax, which facilitates test generation while preserving syntactic correctness. FuzzFlow utilizes a graph-based intermediate representation, which explicitly models control flow and data flow, enabling fine-grained semantic mutations.

Input Generation Strategy. Concerning input generation, three tools rely exclusively on mutation-based strategies, two follow a generational approach based on grammar, and the great majority (14 – 74%) adopt a hybrid strategy that combines both methods to produce new test inputs.

Error detection. Only six tools (31%) include error detection mechanisms that validate the syntax and semantics of generated inputs during preprocessing, ensuring malformed inputs are filtered out.

Code coverage feedback. Nine tools (47%) incorporate code coverage feedback mechanisms to assess how much of the engine's code is exercised during testing, enabling better guidance of the fuzzing process. The use of coverage is controversial in the literature. While many authors consider that a great coverage of statements implies a greater number of revealed faults, many authors argue that some parts of the code are not equally

fault-prone. Considering this, JSTargetFuzzer adopts a history-based approach, leveraging commit history and machine learning techniques to identify areas more related to vulnerabilities and that are targeted in the fuzzing process.

Input Evaluation. Input evaluation, or oracle mechanisms, vary across JavaScript engine fuzzers. Most tools in our survey (13 in total) rely on process-level oracles, such as crashes, assertion failures, or abnormal termination, to detect memory safety violations. In contrast, FuzzJIT employs a semantic oracle based on execution comparison, using input templates that trigger JIT compilation and comparing interpreted and JIT-compiled executions to expose subtle miscompilations without requiring crashes. Similarly, JIT-Picker adopts a differential testing oracle, contrasting executions with and without JIT optimizations to reveal silent semantic deviations. Other tools, including Fuzzilli, JSTarget-Fuzzer, CovRL-Fuzz, and Dumpling, leverage engine instrumentation (e.g., coverage feedback or execution-state signals) to guide exploration; while still crash-confirmed, this instrumentation enables more targeted exploration than purely crash-driven approaches.

4.3. Benchmarks used in evaluation

To reach a benchmark for the area, we also investigated the engines used in the evaluation of the tools. Table 3 presents the JavaScript engines assessed by each fuzzer. There are some engines (namely, TraceMonkey, Hermes, and Espruino) targeted by only one tool/study. V8, JavaScriptCore and ChakraCore are the most popular and used. ChakraCore is the preferred. A possible reason for this is that, besides being highly popular and relevant engines, they have extensive and accessible documentation related to their security patches.

Table 3. Target JavaScript Engines

JS Engine	Tools	References
TraceMonkey	1	[Holler et al., 2012]
Hermes	1	[Groß et al., 2023]
Espruino	1	[Lin et al., 2019]
QuickJS	2	[Wang et al., 2025; Xu et al., 2024a]
Duktape	2	[Groß et al., 2023; Oliveira et al., 2025]
JerryScript	8	[Eom et al., 2024; Groß et al., 2023; Lin et al., 2019; Oliveira et al., 2025; Wang et al., 2019, 2025; Xu et al., 2024a,b]
SpiderMonkey	8	[Bernhard et al., 2022; Mozilla, 2019; Park et al., 2020; Veggalam et al., 2016; Wang et al., 2023, 2025; Xu et al., 2024a,b]
V8	11	[Bernhard et al., 2022; Eom et al., 2024; Groß et al., 2023; Han et al., 2019; Holler et al., 2012; Park et al., 2020; Wachter et al., 2025; Wang et al., 2023, 2025; Xu et al., 2024a,b]
JavaScriptCore	13	[Bernhard et al., 2022; Eom et al., 2024; Groß et al., 2023; Han et al., 2019; Lin et al., 2019; Oliveira et al., 2025; Park et al., 2020; Sun et al., 2022; Wang et al., 2019, 2023, 2025; Xu et al., 2024a,b]
ChakraCore	15	[Bernhard et al., 2022; Eom et al., 2024; Groß et al., 2023; Han et al., 2019; Lee et al., 2020; Lin et al., 2019; Oliveira et al., 2025; Park et al., 2020; Sun et al., 2022; Wang et al., 2017, 2019, 2023, 2025; Xu et al., 2024a,b]

5. Trends and Research Opportunities

During the analysis of the tools, we identified some limitations mentioned by the authors. They are discussed next as well as research opportunities and trends for future investiga-

tion.

Our review shows the tools adopt AST or IL based on grammars. The preprocessing step is important to ensure the generation of only valid inputs, increasing fuzzing performance. The use of grammars allows for the addition of new features in the tool and the rapid delivery of new versions to follow the evolution of JavaScript language.

We also observe that great majority of the tools do not provide targeted mechanisms to focus on specific types of vulnerabilities. Instead, they employ a broad fuzzing strategy, occasionally allowing limited customization during input generation [Han et al., 2019]. Even tools that focus on a particular engine behavior typically do not allow further reconfiguration to explore other vulnerability classes, which restricts their applicability to a narrow scope. More recent tools PatchFuzz and JSTargetFuzzer utilize historical data from commits to guide the fuzzing process. The use of code coverage is an important feature, present in the great majority of tools. But we observe a trend that is to consider that covered parts of the code are not equally associated to vulnerabilities. There are some parts that should be prioritized because are more security-sensitive.

Most fuzzers targeting JavaScript engines rely on static configurations and lack adaptive mechanisms to adjust strategies during execution. This limits their ability to explore deeper or more security-critical code paths, especially in engines with complex and varied runtime behavior [Han et al., 2019; Holler et al., 2012; Lee et al., 2020; Wang et al., 2019]. Some other tools are designed to uncover specific vulnerabilities [Bernhard et al., 2022; Sun et al., 2022; Wang et al., 2023]. Most fuzzers rely on crash-based mechanisms to detect vulnerabilities, which prevents them from identifying non-crashing issues such as memory leaks [Holler et al., 2012]. Even when custom oracles are used to assess individual inputs, existing tools still lack mechanisms to target various vulnerability classes [Bernhard et al., 2022; Wang et al., 2023].

Focusing on more relevant areas is related to performance, an important aspect that needs to be addressed. Fuzzing traditionally consumes significant resources. Only five tools in our review are multithreaded, a characteristic that is essential for making their practical use feasible and that directly impacts performance. Speedup gains are welcome, as code instrumentation (used in several fuzzers) to obtain coverage makes the fuzzing process even slower. Moreover, a large number of inputs needs to be executed, highlighting the need for approaches that reduce or eliminate the number of executions. A characteristic still related to performance is the quality of the initial corpus, which should be highly diverse and contain only valid inputs. Research should concentrate effort in methods to keep corpus small and representative, without redundant inputs which produce known outputs, which lead to a waste of time and resources.

In Section 4.3, we report the main engines used in the evaluation of the tools, but we observe a lack of benchmark to the area, as well as studies comparing the performance of the existing tools. To conduct experiments with this goal is a research opportunity, which could point out the tools limitation and trade-offs.

We also observe a trend that is the use of machine learning to help in the fuzzing process. For instance, mining historical data set, classify error code and kinds of vulnerabilities [Oliveira et al., 2025]. As happen in different testing tasks, it is estimated a crescent number of tools adopting generative techniques, such as *LLMs (Large Language*

Models). These models can improve performance of different fuzzing steps. LLMs can better understand the code and bugs, as well as detect vulnerable parts. There are some studies and LLM-based fuzzers [Jiang et al., 2024], although not focused on JavaScript engines.

6. Concluding Remarks

JavaScript engines remain high-value targets for security attack in modern computing environments. Their widespread use across browsers, servers, mobile platforms, and embedded devices, combined with open-source implementations, development with memory-unsafe languages, and dynamic typing, creates a large and complex attack surface. This complexity makes securing them both essential and challenging.

Fuzzing has emerged as one of the most scalable and cost-effective techniques for identifying vulnerabilities in such engines. Although existing tools have made substantial progress in input generation, seed management, and preprocessing, some areas remain underexplored. To better understand the limitations and opportunities in fuzzers for JavaScript engines, we examined 19 state-of-the-art tools. We noticed that these tools employ diverse methods for input generation, ranging from mutation to grammar-based synthesis.

Although each tool has its technical peculiarities, they share common features that helped us identify an important gap. Most tools adopt a general-purpose approach for discovering bugs without targeting specific vulnerability types. Others, while specialized, are often restricted to a narrow class of issues [Bernhard et al., 2022; Sun et al., 2022; Wang et al., 2023] and lack flexibility for broader exploration [Wang et al., 2025]. This observation highlights the need to implement a more adaptable and targeted fuzzing strategy.

Acknowledgments

This work is partially supported by FAPESP (grant #2023/00577-8), CNPq (grants #310034/2022-1, #444311/2024-6 and #406941/2025-4), and CAPES-PROEX.

References

Lukas Bernhard, Tobias Scharnowski, Moritz Schlögel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential fuzzing of JavaScript engines. In *ACM Conference on Computer and Communications Security CCS*, pages 351–364. ACM, 2022.

Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.02.002. URL `https://www.sciencedirect.com/science/article/pii/S0167404818300658`.

Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. *ACM Comput. Surv.*, 56(9), April 2024. ISSN 0360-0300. doi: 10.1145/3648468. URL `https://doi.org/10.1145/3648468`.

I Putu Arya Dharmaadi, Elias Athanasopoulos, and Fatih Turkmen. Fuzzing frameworks for server-side web applications: a survey. *International Journal of Information Security*, 24(2):73, 2025.

Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your javascript: hygienic macros for ES5. In *DLS*, pages 35–44. ACM, 2014.

Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. Fuzzing JavaScript interpreters with coverage-guided reinforcement learning for LLM-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, pages 1–13, New York, NY, USA, September 2024. ACM. doi: 10.1145/3650212.3680389. URL https://doi.org/10.1145/3650212.3680389.

Google. american fuzzy lop. https://github.com/google/AFL, 2023. Accessed in 08/24/2022.

Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for javascript jit compiler vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium 2023*, pages 10–25, San Diego, CA, USA, 2023.

HyungSeok Han, DongHyeon Oh, and Sang Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Network and Distributed System Security Symposium*, 01 2019. doi: 10.14722/ndss.2019.23263.

Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association. ISBN 978-931971-95-9. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 492–496, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706585. doi: 10.1145/3663529.3663784. URL https://doi.org/10.1145/3663529.3663784.

Zeyan Kang. A review on javascript engine vulnerability mining. In *Journal of Physics: Conference Series*, volume 1744, page 042197. IOP Publishing, 2021.

Holger M. Kienle. It's about time to take JavaScript (more) seriously. *IEEE Software*, 27 (3):60–62, 2010. doi: 10.1109/MS.2010.76.

Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung.

J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(6), 2018.

Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing bugs in JavaScript engines through test transplantation and differential testing. *Software Quality Journal*, 29(1):129–158, 2021.

Hongyang Lin, Junhu Zhu, Jianshan Peng, and Dixia Zhu. Deity: Finding deep rooted bugs in JavaScript engines. In *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, pages 1585–1594, 2019. doi: 10.1109/ICCT46805.2019.8947153.

Sanoop Mallissery and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Computing Surveys*, 56(3):1–38, 2023.

Tommi Mikkonen and Antero Taivalsaari. Using JavaScript as a real programming language. Technical report, Sun Microsystems, Inc., USA, 2007.

Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

Mozilla. jsfunfuzz. `https://github.com/MozillaSecurity/funfuzz`, 2019.

Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. A machine learning approach to detection of JavaScript-based attacks using ast features and paragraph vectors. *Applied Soft Computing*, 84:105721, 2019.

Bruno G Oliveira, Andre T. Endo, and Silvia Vergilio. Using historical information for fuzzing JavaScript engines. In *In Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS)*, volume 2, pages 59–70, 2025. ISBN 978-989-758-749-8.

Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, 2020. doi: 10.1109/SP40000.2020.00067.

Marcus Rodrigues, Breno Guimarães, and Fernando Magno Quintão Pereira. Generation of in-bounds inputs for arrays in memory-unsafe languages. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 136–148. IEEE, 2019.

J. Ruderman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`, 2007.

Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, San Francisco, CA, USA, 2015.

Lili Sun, Chenggang Wu, Zhe Wang, Yan Kang, and Bowen Tang. KOP-Fuzzer: A key-operation-based fuzzer for type confusion bugs in JavaScript engines. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 757–766, 2022. doi: 10.1109/COMPSAC54236.2022.00125.

Ye Tian, Xiaojun Qin, and Shuitao Gan. Research on fuzzing technology for JavaScript Engines. In *Proceedings of the 5th International Conference on Computer Science and Application Engineering*, pages 1–7, 2021.

Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 581–601, Cham, 2016. Springer International Publishing. ISBN 978-3-319-45744-4.

Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. Dumpling: Fine-grained differential javascript engine fuzzing. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, 2025.

Jie Wang. Characterizing and taming non-deterministic bugs in javascript applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1006–1009, 2017. doi: 10.1109/ASE.2017.8115720.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017. doi: 10.1109/SP.2017.23.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019. doi: 10.1109/ICSE.2019.00081.

Junjie Wang, Zhiyi Zhang, Qi An Xin, Shuang Liu, Xiaoning Du, and Junjie Chen. FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity23/presentation/wangjunjie.

Junjie Wang, Yuhan Ma, Xiaofei Xie, Xiaoning Du, and Xiangwei Zhang. Patchfuzz: Patch fuzzing for javascript engines, 2025. URL https://arxiv.org/abs/2505.00289.

Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. Fuzzing javascript engines with a graph-based ir. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3734–3748, New York, NY, USA, 2024a. Association for Computing Machinery. ISBN 9798400706363. doi: 10.1145/3658644.3690336. URL https://doi.org/10.1145/3658644.3690336.

Haoran Xu, Yongjun Wang, Zhiyuan Jiang, Shuhui Fan, Shaojing Fu, and Peidai Xie. Fuzzing javascript engines with a syntax-aware neural program model. *Computers and Security*, 144:103947, 2024b. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2024.103947. URL https://www.sciencedirect.com/science/article/pii/S0167404824002529.

Zhenhua Yu, Zhengqi Liu, Xuya Cong, Xiaobo Li, and Li Yin. Fuzzing: Progress, challenges, and perspectives. *Computers, Materials and Continua*, 78(1):1–29, 2024. ISSN 1546-2218. doi: https://doi.org/10.32604/cmc.2023.042361. URL https://www.sciencedirect.com/science/article/pii/S154622182400170X.

Man Zhang and Andrea Arcuri. Open problems in fuzzing RESTful APIs: A comparison of tools. *ACM Trans. Softw. Eng. Methodol.*, 32(6), September 2023. ISSN 1049-331X. doi: 10.1145/3597205. URL https://doi.org/10.1145/3597205.

Xiaoqi Zhao, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv, and Gai-Ge Wang. A systematic review of fuzzing. *Soft Computing*, 28(6):5493–5522, 2024.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022. ISSN 0360-0300. doi: 10.1145/3512345. URL https://doi.org/10.1145/3512345.