Research on Fuzzing Technology for JavaScript Engines

Ye Tian
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zheng zhou, China Wuxi, China
sweetwild@qq.com

Xiaojun Qin
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
xjqin@163.com

Shuitao Gan*
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
ganshuitao@gmail.com

ABSTRACT

JavaScript engine is the core component of web browsers, whose security issues are one of the critical aspects of the overall Web Eco-Security. Fuzzing technology, as an efficient software testing approach, has been widely applied to detecting vulnerabilities in different JavaScript engines, which is a security research hotspot at present. Based on systematical dissection of existing fuzzing methods, this paper reviews the development and technical ideas of *JavaScript Engine Fuzzing* combined with taxonomy, proposes a general framework of *JavaScript Engine Fuzzing* and analyzes the key techniques involved. Finally, we discuss the core issues that restrict efficiency in current research and present an outlook on the future trends of *JavaScript Engine Fuzzing*.

CCS CONCEPTS

• **Security and privacy**; • **Software and application security**; • **Software security engineering**;

KEYWORDS

Browser security, JavaScript engine, Fuzzing, Vulnerability detection

ACM Reference Format:
Ye Tian, Xiaojun Qin, and Shuitao Gan. 2021. Research on Fuzzing Technology for JavaScript Engines. In *The 5th International Conference on Computer Science and Application Engineering (CSAE 2021), October 19–21, 2021, Sanya, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3487075.3487107

1 INTRODUCTION

As the interactive window between users and the information network, browser is an indispensable part of the web-ecosystem, and also one of the tempting targets for cyber-attacks. Internet giants, including Google, Microsoft and Apple, need to spend plenty of manpower and resources to research and protect the security of their browsers every year. JavaScript engine is the central component of the browser kernel, which mainly supports the dynamic interaction of pages by interpreting and executing JS scripts in real time. At present, more than 97.4% of all websites use JavaScript for client-side programming [1], hence JavaScript engines are attractive targets for attackers.

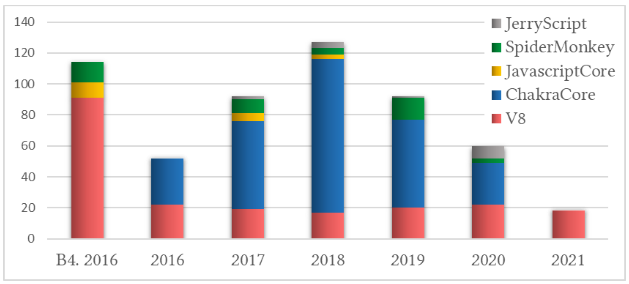

Figure 1: Vulnerability Exposure Statistics for Mainstream JavaScript Engines.

To ensure efficient parsing and execution of JS scripts, JavaScript engines are usually designed and implemented based on C/C++, making them at higher risk of security threats and vulnerable to malicious exploitation by attackers. Through a cross-comparison of the NVD(National Vulnerability Database) and CVE(Common Vulnerabilities and Exposures) databases, there are more than 500 medium/high risk vulnerability entries exposed to V8 [2], ChakraCore [3], JavaScriptCore [4], SpiderMonkey [5], JerryScript [6] and other mainstream JavaScript engines , among which ChakraCore and V8 both have more than 200 entries (shown in Figure 1). Attackers can indirectly exploit JavaScript engine vulnerabilities to cause remote command execution and gain control of the system, jeopardizing the user information and host security. In this context, it is important to employ fast and effective software testing techniques to analyze JS engines and find flaws in the design, development and optimization process.

Fuzzing (also called Fuzzy testing) [7] is a well-known technique in vulnerability detection, especially for software like JavaScript engines, which have large code amounts and complex structures. Fully taking the advantages of automated testing, fuzzing can notably reduce the dependence of prior knowledge and labor costs. Moreover, Coverage Feedback, Deep Learning, Distributed Computing and other techniques are adopted to enrich the fuzzing framework, which improves the vulnerability detection capacity towards JavaScript engines. In terms of development process, *JavaScript Engine Fuzzing* has evolved from manually redacting grammar templates for test case generation to automatically constructing test cases with features such as diverse code mode and cross-platform support.

In this paper, we try to review the development of *JavaScript Engine Fuzzing*, summarize the corresponding fuzzing techniques in

CSAE 2021, October 19–21, 2021, Sanya, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8985-3/21/10…$15.00
https://doi.org/10.1145/3487075.3487107

combination with the operating mechanisms and vulnerability detection principles of JS engines, and tease out the challenges. Based on comprehensive research of methodology and technical implementation for *JavaScript Engine Fuzzing*, we classify and generalize the mainstream *JavaScript Engine Fuzzers* (fuzzing tools) according to test-case construction, and propose a general *JavaScript Engine Fuzzing Framework*. Then, we further analyze the key techniques involved and the problems to be solved in *JavaScript Engine Fuzzers*, and point out the factors that restrict fuzzing efficiency and the corresponding solutions. Finally, we present the conclusion and outlook on *JavaScript Engine Fuzzing* techniques in line with practical analysis.

2 OVERVIEW

JavaScript is a scripting language that supports page dynamic interaction, which is broadly utilized in web development related areas. JavaScript engine is a kind of component program that dynamically parses and executes JS scripts according to the ECMAScript standard [8], which can be regarded as a dedicated script processing virtual machine. Its core function is to provide dynamic interaction capabilities for applications such as browser that support the JavaScript language API. In essence, JavaScript engine is a driver that converts JS code into machine code and executes it.

2.1 JS Engine Operating Mechanism

JavaScript engine usually consists of four parts: parser, interpreter, JIT compiler and runtime environment. The dynamic parsing of JS scripts by JavaScript engine can be divided into two phases: code validity check and code interpreting execution. The validity check stage includes syntax analysis and semantic analysis, while the interpreting execution stage includes pre-parsing and actual execution. The specific operating principle is shown in Figure 2 JavaScript engine first uses the syntax parser to parse the JavaScript code into the Abstract Syntax Tree (AST), and then the interpreter analyzes the AST to generate the byte-code based on line by line code interpretation. To maintain fast execution of the JS code, the runtime environment monitors the running code and then feeds optimizable code to the JIT (Just-In-Time) compiler for optimization, which generates optimized machine code to replace the bytecode not optimized by previous interpreter. Finally, the optimized machine code is executed by JavaScript engine to render corresponding interactive functions of the script. What calls for special attention is that JavaScript engine will run the unoptimized bytecode before the code is compiled for optimization, thereby ensuring the engine's actual processing efficiency.

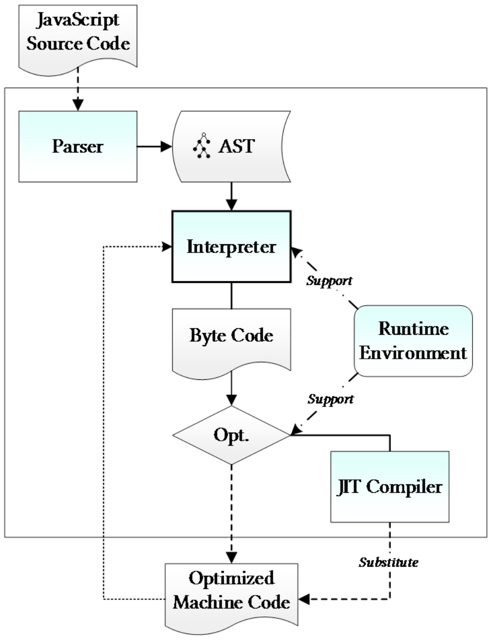

Figure 2: Schematic of JS Engine Operating Mechanism.

2.2 JS Engine Vulnerability Detection Methods

The core principle of JavaScript engine vulnerability detection is to search for input sets that can trigger exceptions or crashes in the input space following the corresponding rules, and find the specific user-mode input that matches the vulnerability profile. According to the different generation mechanisms and internal logic, JavaScript engine vulnerabilities can be categorized into four types, namely Overflow, Use After Free, Race Condition and Type Confusion vulnerabilities [9].

Due to the large code amount and complex functional structure of JavaScript engines, the manual analysis approach requires heavy effort in source code audit and reverse analysis, leading to inefficient vulnerability detection. Therefore, the industry mostly applies static analysis and fuzzing to detect vulnerabilities in JavaScript engines with less reliance on manual work.

Static analysis [10] can automatically analyze the program execution paths without running the application under test by introducing intermediate representations, formal logic and other techniques to detect potential flaws arising from the design and development process. Typical static analysis tools [11-13] that leverage data flow analysis are sensitive to programming languages implementing JavaScript engines, such as C/C++, and can be applied to JS engine vulnerability detection, but the uptilted false alarm rate may greatly reduce the credibility of static analysis reports. In addition, static analysis methods based on taint analysis [14, 15] and symbolic execution [16, 17] theoretically have the capacity to analyze JavaScript engines with higher accuracy. However, conditioned by the complex logic structure and dynamic execution mechanism of JS engines, the actual testing process may have low execution speed and is susceptible to memory explosion, path explosion, etc., which seriously restricts the vulnerability detection capability of the static analysis. Currently, studies [18-20] using static analysis to detect vulnerabilities in JavaScript engines mainly focus on how to reduce manual involvement and test running overhead.

In comparison, fuzzing pays more attention to the automation and execution speed during the detection process, and requires less artificial expertise and analysis, making it more suitable for detecting vulnerabilities in intricate JavaScript engines.

2.3 Fuzzing Technique Outline

Fuzzing [7, 21] is a software testing technique that automatically constructs and inputs unexpected data in accordance with the characteristics of target objects, while monitoring operational anomaly signals to detect potential vulnerabilities. With simple design & deployment, low operational overhead, nice scalability and high reliability, fuzzing has been unanimously accepted by industry and academia, thus commonly used in software security fields. SAGE [22], honggfuzz [23], AFL [24], libFuzzer [25] and other fuzzers launched by security teams of Internet vendors are deployed for security tests throughout the lifecycle of software development & maintenance, and have identified quite a few undisclosed vulnerabilities in various applications/systems.

After years of technical accumulation, fuzzing has evolved from the early random testing to the stage of targeted and oriented testing in line with the characteristics of the PUT (program under test). In terms of the input data type used by PUTs, there are mainly file-based, network-protocol-based, kernel-based and API-based fuzzers. *JavaScript Engine Fuzzing* requires to input JS code generated under certain rules into the JavaScript engines, which is a typical application of file-based fuzzers, with early representative tools such as Peach [26] and FileFuzz [27]. Fuzzing can be categorized into black-box, white-box and grey-box models according to the dependency of PUTs' internal knowledge structure. Since JavaScript is a dynamic weakly typed language, the white-box model cannot meet the needs of dynamic execution, so *JavaScript Engine Fuzzing* mainly adopts black-box or grey-box models. In practice, fuzzing technology can detect various kinds of vulnerabilities, and has become the mainstream JavaScript engine vulnerability detection means.

2.4 Challenges

Unlike ordinary software, JavaScript engines have complex logic and huge code scale, increasing the difficulty of vulnerability detection. Compared with general fuzzing methods, efficient *JavaScript Engine Fuzzing* needs to exceed the following issues.

2.4.1 Test-Case Validity Check. JavaScript engines have a strict validity check mechanism in the JS script parsing process (shown in Figure 3). When encountering code that does not conform to the syntax or semantic specifications, the check mechanism will terminate the parsing of the entire test case, which prevents the test case from touching the core functional module of JavaScript engines and makes it difficult to achieve the ideal testing results. If we blindly apply the random mutation strategy to generating test cases, the legitimacy of highly structured inputs such as JS code would be undermined with great ease. As a result, the fuzzing process may consume plenty of time in the early parsing phase, making it difficult to find much more valuable deep-rooted bugs, which in turn limits the efficiency of detecting vulnerabilities in JavaScript engines.

2.4.2 Code Coverage Enhancement. Code coverage capability is one of the vital indicators to measure the vulnerability detection ability of a fuzzing tool. Better code coverage capability means a higher probability of detecting valid vulnerabilities. In order to

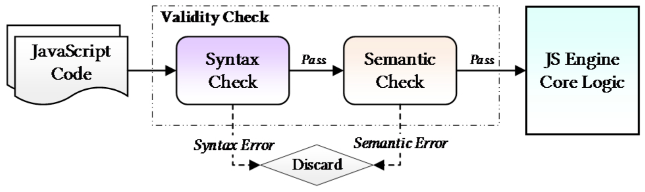

Figure 3: JavaScript Engine Validity Check Mechanism.

implement multiple Web dynamic interaction functions and ensure browser's reliable operation, the mainstream JavaScript engines have a large amount of code, among which the smallest one, JavaScriptCore, has 450,000 lines of critical code, while SpiderMonkey has more than one million lines. The enormous code scale makes it impossible to effectively fuzz most of the program, and test cases can only cover limited execution paths. Therefore, how to construct test cases with higher code coverage is a great challenge for fuzzing JavaScript engines.

3 CLASSIFICATION

Test cases are pivotal to the capability of discovering JavaScript engines' vulnerability. Consequently, unlike traditional software testing techniques centered on the PUTs, *JavaScript Engine Fuzzing* can be classified into *Generation-Based JS-Engine Fuzzing*, *Mutation-Based JS-Engine Fuzzing* and *Composite (generation + mutation) JS-Engine Fuzzing* around the test case construction.

3.1 Generation-Based JS-Engine Fuzzing

Generation-based fuzzing is also called model-based fuzzing [28], as it generates ideal test cases by constructing specific models based on the analysis of the PUTs' expected inputs. Applied to the JavaScript engine vulnerability detection, the crux of generation-based fuzzing lies in how to automatically construct test cases with rich code-style and JS-specification compliance.

Early generation-based fuzzers such as Peach [26] required to manually construct test templates based on the characteristics of JavaScript engines, which generated limited samples with poor validity. JSfunfuzz [29] introduced JS grammar templates, and for the first time used grammar-based input generation, which was able to generate random and syntactically correct test cases and found quantities of JavaScript engine vulnerabilities. However, to guarantee the test cases' validity, researchers were required to expend massive efforts to construct grammar rule templates, and the automation level of JSfunfuzz is relatively low.

In order to enhance the automation of fuzzing, TreeFuzz [30] and Skyfire [31] learned grammatical features and rules from existing samples to generate JS test inputs by introducing probabilistic models. TreeFuzz leveraged probabilistic context free grammar (PCFG) to automatically infer test case generation models from the given JS corpus, while Skyfire adopted probabilistic context sensitive grammar (PCSG) to generate a much more reasonable distribution of test cases. Both two fuzzers employed the idea of data-driven test case generation, which overcame the reliance on prior knowledge. CodeAlchemist [32] proposed a semantics-aware assembly technique, which first performed initial seed fragmentation using abstract syntax tree (AST), and then fetched the variable definitions according to control flow and data flow analysis after preprocessing measures such as de-duplication and variable normalization, and then identified the variable types in combination with code instrumentation. Finally, code fragments matching constraints were assembled to generate semantically correct JS test cases. This semantics-aware split-combination approach could effectively increase the test case legitimacy and reduce the invalid test overheads arising from runtime errors.

Montage [33] pioneered the application of the neural network language model (NNLM) to fuzzing JavaScript engines, utilizing JS code fragments filtered and processed from the Test262 [34] dataset to train the LSTM (Long Short Term Memory) generation model. The LSTM model learned the syntax, semantic patterns and control flow features of JavaScript training set by deducing correlations between code fragments, which markedly decreased the possibility of triggering runtime errors. Comfort [35] replaced the NNLM model with an advanced Transformer-based GPT-2 model, which utilized differential testing and language specification to assist fuzzing, further lifting the efficiency and effectiveness of test case generation. The JS test cases generated by DL models have abundant code-styles and higher accuracy, while the fuzzers' code coverage and vulnerability detection capability have noticeable improvement.

3.2 Mutation-Based JS-Engine Fuzzing

Mutation-based fuzzing constructs new test cases by mutating initial inputs through random or heuristic strategies, free from the dependence on generative models, also known as model-less fuzzing [28]. Compared to generation-based fuzzers, mutation-based fuzzers execute faster, and further improve code coverage due to the inclusion of feedback mechanism. AFL [24] is the most representative mutation-based fuzzer in recent years, and several works [36-38] have optimized and extended upon AFL to explore numerous unrevealed vulnerabilities. However, owing to the JavaScript engines' strict input validity check mechanism, it is difficult for JS samples to reach the core logic of JS engines if the byte/bit-level random mutation of test cases is directly used by fuzzers such as AFL, resulting in inefficient fuzzing. Therefore, mutation-based *JavaScript Engine Fuzzing* necessitates taking full account of the JavaScript language structural features with a primary focus on what level of mutation is conducted.

Superion [39] built a grey-box model that performed mutation at the JavaScript AST level based on AFL and ANTLR [40], using a syntax-aware trimming strategy to directly reduce the test input space at AST level, and then utilizing subtree replacement to achieve syntax-aware seed mutation. This approach dramatically reduced the invalidity of JS test cases solely produced by AFL. Under the same conditions, the code coverage and vulnerability detection capability of Superion distinctly surpassed the previous fuzzing methods. Similarly, Deity [41] assisted mutation process with feature templates extracted from 1-day exploit samples and employed a more analytically efficient tool, Esprima, to realize operations such as trimming and mutation towards the AST. SaFuzzer [42] further incorporated a semantic-repair mechanism to solve the syntactically valid but semantically invalid problem of test cases with mutated output at AST level, reducing the chance of triggering runtime errors and thus improving the vulnerability detection efficiency.

Fuzzilli [43] introduced a custom intermediate language, FuzzIL, which converted JS code into FuzzIL that is closer to the actual bytecode executed by JavaScript engine, and reverted to JS code (the newly constructed test cases) after mutation operations at the intermediate language level. Mutations at IL level effectively inherited the initial inputs' control flow and data flow attributes, thus enabling the semantic validity of test cases. In comparison with the AST-based mutation methods, however, Fuzzilli ignored part of the path coverage information in the structure design and was less capable of spotting bugs in the JIT optimization process.

3.3 Composite JS-Engine Fuzzing

Generative-based fuzzing can strictly control the generation of each test case statement, resulting in high test validity, but also produce a huge test input space. Mutation-based fuzzing is more efficient, but the validity of test cases generated by mutation strategy is relatively low. To balance test validity and efficiency, most *JavaScript Engine Fuzzzer*s combine the ideas of generation and mutation, i.e., composite JS engine fuzzing.

LangFuzz [44] was the earliest one to apply code-fragmentation ideas to fuzzing interpreters and pioneered the composite JS-engine fuzzing. LangFuzz parsed legitimate JavaScript samples with AST, learned to extract various code fragments and stored them as token streams in a fragments pool, and then replaced the fragments to build new test cases. Test cases were constructed in both generation and mutation means (mutation dominating), where the generative one leveraged breadth-first strategy to replace non-terminal AST nodes, and the mutative one leveraged random strategy to replace the fellow fragments. GramFuzz [45] adopted depth-first strategy to traversal search each node of AST and performed mutations such as deletion, modification and duplication with the adjustable probability parameter $\boldsymbol{P}$. IFuzzer [46] introduced the genetic programming that used genetic algorithms and evolutionary strategies to select, crossover and mutate JS samples, further extending the test case construction mode. LangFuzz, GramFuzz and IFuzzer all use context-free grammar to generate and mutate code fragments at AST level, vastly elevating JS test case construction. Nevertheless, confined by the absence of feedback mechanism in black-box model, they were prone to affect the efficiency of fuzzing JavaScript engines due to excessive randomness.

Nautilus [47] constituted a grey-box JS engine testing model by adding instrumentation and coverage feedback techniques based on generative fuzzer. It first instrumented on the source code in an AFL-like manner, and then generated inputs for mutation at the syntax tree level based on grammar rules. After refinement of the input, Nautilus combined the feedback information to select the suitable input mutation method from five different mutation strategies and constructed new JS test cases.

DIE [48] proposed an Aspect-Preserving mutation technique to retain special structure and type information that might trigger vulnerabilities during JS test cases construction, addressing the issue that code-fragmentation might corrupt the logical structure and subtle semantic of original seeds. Based on the high-quality

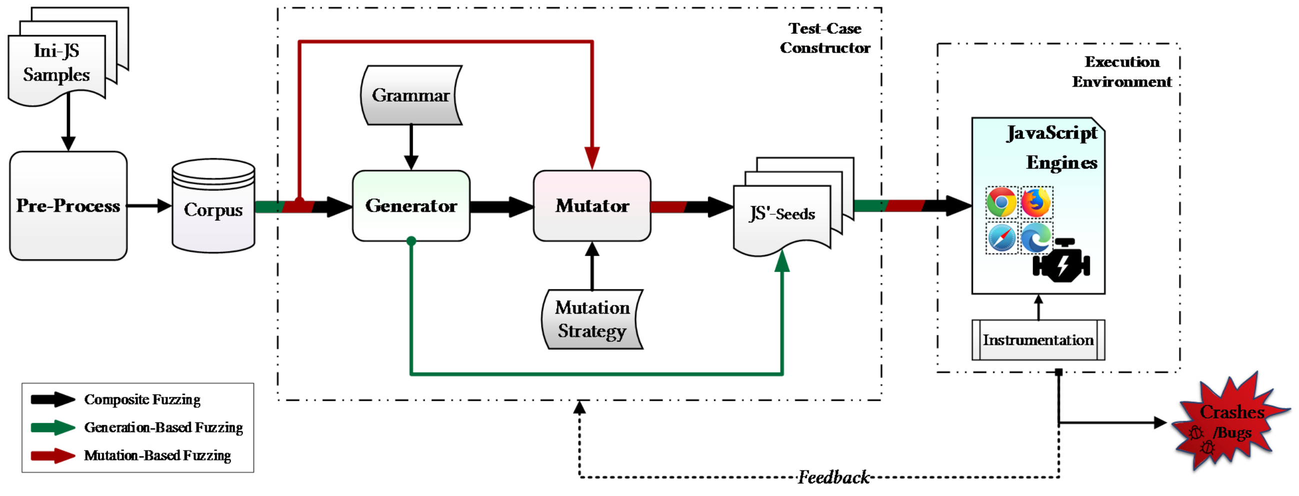

Figure 4: JavaScript Engine Fuzzing Framework.

JS vulnerability samples (including PoCs and test suites), DIE generated initial corpus through lightweight type analysis, and then constructed new test cases with Structure-preserving and Type-preserving mutation strategies. The entire process centered on so-called *Aspect*, namely elements with high probability of triggering vulnerabilities, preserving the critical code structures that affected the control flow and data dependency information as much as possible. The validity of test cases and the ability to locate deep-rooted bugs (e.g., JIT optimization procedure defects) have been significantly improved. At the same time, combined with coverage feedback technique, DIE can effectively shrink the JavaScript engines' test input space and lessen invalid mutations, which is the best *Composite JS-Engine Fuzzer* in practice.

4 FUZZING FRAMEWORK

JavaScript Engine Fuzzing Framework is analogous to the usual fuzzing framework which requires various design aspects such as corpus pre-processing, test case construction and execution environment for target engines. With the discussion in **Section 3**, the overall structure of different *JavaScript Engine Fuzzers* can be summed up in the general framework illustrated in Figure 4, despite variance in test procedures and design implementation details. Firstly, the pre-processing module pre-processes initial JS samples and stores them in the corpus. The test case constructor then processes corpus in generation or mutation or composite means to construct JS seeds (i.e. test cases). Finally, the execution environment feeds the JS seeds into pre-instrumented JavaScript engines to execute and throws crashes or exceptions while running. Moreover, during JS seed execution, fuzzers who have feedback mechanisms provide guided feedback on test cases construction together with coverage information.

Among them, test case constructor is the core of framework, having the greatest impact on the effectiveness of *JavaScript Engine Fuzzing*. Meanwhile, to perform efficient vulnerability detection, we also require attention to initial corpus construction, effectiveness & efficiency balance, and runtime feedback.

4.1 Initial Corpus Construction

Initial corpus is the igniter of fuzzing. High-quality initial corpus enables high code coverage at the start of fuzzing, which is vital to the performance of fuzzers. The JS samples for constructing initial corpus are mainly derived in four ways.

i. Manually written by testers
ii. Crawling from web pages
iii. Available PoCs and test suites
iv. Official test datasets

The former two suffer from an irreconcilable conflict between test orientation and cost, so most current fuzzers adopt the latter two ways to obtain JS initial samples. In addition, mainstream *JavaScript Engine Fuzzers* construct test cases at AST level and require pre-processing to convert initial corpus into AST form. DIE [48] developed a lightweight type analyzer that transformed public vulnerability samples into Typed-AST and deposits them in the corpus. Montage [33] leveraged Deep Learning model for adaptive preprocessing based on the official ECMA test suites.

4.2 Effectiveness & Efficiency Balance

The crux of fuzzing is how to reduce the input space and build optimized test cases for accurate and efficient vulnerability detection. To improve effectiveness, differential testing [49] and language specification assistance [35] have been applied to the design of *JavaScript Engine Fuzzers*. Fuzzilli [43] and DIE [48] introduced distributed technology to improve fuzzing efficiency. Some commercial firms have also employed large-scale clusters like ClusterFuzz [50] and OSS-Fuzz [51] to enable more efficient vulnerability detection. However, restricted by the complex logic structure and large code size of JavaScript engines, it is hard to satisfy both efficiency and accuracy in fuzzing, which requires a balance between them in line with the fuzzer design principles. Taking DIE as an example, instead of traversing all the valuable code structures to pursue validity, a random retention strategy is selected to elevate the overall fuzzing efficiency. In general, the generation-based approach focuses on test cases' validity, while the mutation-based approach cares more about efficiency.

4.3 Runtime Feedback Techniques

Feedback mechanism is also essential for the fuzzing framework, mainly relying on instrument and coverage feedback. Fuzzers applying feedback techniques [39, 41-43, 47, 48] can reduce the overhead of invalid test case generation, explore more valuable program execution paths and improve the efficiency of detecting vulnerabilities in JS engines. Most of the JavaScript engines are open-sourced and can insert the probes for coverage feedback directly at the source code level during compiling. For the few JavaScript engines that are closed source, the code coverage can be monitored with the AFL-Qemu model [24] or binary dynamic instrumentation [52, 53]. Alternatively, the Intel-PT mechanism [54] can track control flow information while program executing at the hardware level, allowing efficient coverage feedback to be obtained without source code.

5 SUMMARY & OUTLOOK

As functionality increases and performance improves, JavaScript engines' software code size and inherent logic complexity are growing rapidly. Therefore, it is a hot topic in security research to locate JS engine vulnerabilities more precisely and efficiently and to patch them quickly according to the threat level. Fuzzing, as an efficient tool to detect JavaScript engine vulnerabilities, has been broadly applied in industry and academia, and has achieved considerable results. Employing the NNLM to generate test cases, Montage found 37 unknown bugs in V8, JavaScriptCore and ChakraCore. DIE took full advantage of semantic information and structure features in existing vulnerable samples and mined 48 bugs in the three engines mentioned above. The latest fuzzer, Comfort, can fuzz nine JS engines, including engines in mobile and embedded devices, and identified 158 separate bugs. Notably, JSfunfuzz and LangFuzz have respectively uncovered 2800+ and 2300+ vulnerabilities in SpiderMonkey since inception, vastly enhancing the security of JavaScript engines. Table 1 sums up the 16 *JavaScript Engine Fuzzers* mentioned all above in timeline, comparing and presenting them in six specific ways.

The core and challenge of JavaScript Engine Fuzzing is to build and run high quality test cases efficiently. The mainstream is to carry out corresponding mutation/generation operations at AST level according to predefined models and strategies. In practice, the main constraint on the vulnerability detection efficiency is the complexity of the JavaScript engine, which has two aspects. On the one hand, JavaScript Engine Fuzzing requires more test cases that can trigger deep-rooted bugs. JS engine vulnerabilities have been gradually extended from simple parsing and memory corruption bugs to deep logic optimization bugs, which demand complex conditions to trigger potential weak points, thus placing high requests on the test case validity and execution path depth. On the other hand, the increasing of JS engine code size leads to the outsize of fuzzing search space. Therefore, more test inputs are needed to cover the main execution path of JS engines effectively, which puts higher demands on the fuzzers' operating mechanism and hardware device overhead.

Table 1: Comparison of JavaScript Engine Fuzzers

Fuzzer	***Year***	***T.***	***M.***	***I.C***	***G.A***	***S.A***	***O.S***
jsfunfuzz	2007	G	●		√		√
LangFuzz	2012	M/G	●	√	√		
AFL	2013	M	◐				√
GramFuzz	2013	M/G	●	√	√		
IFuzzer	2016	M/G	●	√	√	√	√
TreeFuzz	2016	G	●	√	√		
Skyfire	2017	G	●	√	√		√
Fuzzilli	2018	M	◐		√	√	√
CodeAlchemist	2019	G	●	√	√	√	√
Superion	2019	M	◐	√	√		√
Deity	2019	M	◐	√	√		
Nautilus	2019	M/G	◐	√	√		√
Montage	2020	G	●	√	√	√	√
SaFuzzer	2020	M	◐	√	√	√	
DIE	2020	M/G	◐	√	√	√	√
Comfort	2021	G	●	√	√	√	√

T: Type(G: generation, M: mutation), ***G.A:*** grammar-aware,
M: Model(●: black-box, ◐: grey-box), ***S.A:*** semantic-aware,
I.C: input-corpus, ***O.S:*** open-source

Currently, generation-based fuzzing applying Deep Learning and feedback-driven composite fuzzing are the two most effectual means of detecting bugs in JavaScript engines. With continuous maturity of artificial intelligence, using neural network models to construct test cases has become a new trend in the *JavaScript Engine Fuzzing*. NEUZZ [55] interacted the DL training process and fuzzing execution process through a socket API, and guided test case generation with edge coverage rate. This idea can be transferred to detect bugs in JavaScript engines. Foreseeably, the integration of feedback-driven gray-box models and DL will become a worthwhile direction for *JavaScript Engine Fuzzing*. In addition, as the potential vulnerability locus in JS engines gradually transition from parser and interpreter to the deeper JIT compiler, it is also an ideal solution to fuzz JIT compiler directly, and a preliminary attempt has been made by JITFuzz [56]. The shift from testing breadth to testing depth will also be a future trend. In the long run, with the development of parallel and distributed computing, it is valuable to explore how to expand the hardware execution capability of *JavaScript Engine Fuzzing* by deploying high-performance computing.

6 CONCLUSION

The security of JavaScript engines is one of the vulnerable elements threatening the whole web ecosystem. Fuzzing, as an efficient and convenient scenario, is currently the most effective approach to detect vulnerabilities in JavaScript engines. In this paper, we try to provide a relatively comprehensive review of the development and current state of *JavaScript Engine Fuzzing*. Combining specific features of JavaScript engines, we analyzed the classification, technical principles, and challenges of previous research, and proposed a general fuzzing framework. Then, we emphatically discussed the cruxes affecting effectiveness and efficiency. At last, we summarized the achievements of typical *JavaScript Engine Fuzzers*, and the potential research direction of future *JavaScript Engine Fuzzing*.

ACKNOWLEDGMENTS

Thanks for the favorable research environment provided by State Key Laboratory of Mathematical Engineering and Advanced Computing. This work is supported by the National Science and Technology Major Projects of HEGAOJI (2018ZX01028102).

Abbreviations

JS: JavaScript; DL: Deep Learning; PoC: Proof of Concept

REFERENCES

[1] W3Techs. Usage Statistics of Javascript for Websites. https://w3techs.com/technologies/details/cp-javascript.
[2] Google. V8: Google's Open Source High-Performance JavaScript and WebAssembly Engine. https://v8.dev/.
[3] Microsoft. ChakraCore: The Core Part of the Chakra JavaScript Engine that Powers Microsoft Edge. https://github.com/microsoft/ChakraCore.
[4] Apple. JavaScriptCore: The Built-In JavaScript Engine for WebKit. https://trac.webkit.org/wiki/JavaScriptCore.
[5] Mozilla. SpiderMonkey: The JavaScript Engine for Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[6] Samsung. Jerryscript: JavaScript Engine for the Internet of Things. https://github.com/jerryscript-project/jerryscript.
[7] B.P. Miller, L. Fredriksen, B. So (1990). An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, 33(12), 32-44.
[8] Ecma-International. ECMAScript®2021 Language Specification. https://www.ecma-international.org/ecma-262/.
[9] H. Lin, J. Peng, S. Zhao, *et al.* (2019). Survey On JavaScript Engine Vulnerability Detection. Computer Engineering and Applications, 55(11), 16-24.
[10] N. Nagappan, T. Ball (2005). Static Analysis Tools as Early Indicators of Pre-Release Defect Density. Proc of the 27th International Conference on Software Engineering, ICSE'05, 580-586.
[11] Synopsys. Coverity Scan Static Analysis. https://scan.coverity.com.
[12] CyberRes. Fortify Static Code Analyzer. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer.
[13] Perforce. Klocwork: Best Static Code Analyzer for Developer Productivity, SAST, and DevOps/DevSecOps. https://www.perforce.com/products/klocwork.
[14] D.E. Denning (1976). A Lattice Model of Secure Information Flow. Communications of the ACM, 19(5), 236-243.
[15] S. Gan, C. Zhang, P. Chen, *et al.* (2020). GREYONE: Data Flow Sensitive Fuzzing. Proc of the 29th USENIX Security Symposium, USENIX Security'20, 2577-2594.
[16] J.C. King (1976). Symbolic Execution and Program Testing. Communications of the ACM, 19(7), 385-394.
[17] R. Baldoni, E. Coppa, D.C. D'Elia, *et al.* (2018). A Survey of Symbolic Execution Techniques. ACM Computing Surveys, 51(3), 50.
[18] C. Omar, J. Aldrich (2016). Programmable Semantic Fragments: The Design and Implementation of Typy. Proc of the ACM SIGPLAN Conference on Generative Programming: Concepts and Experiences, GPCE'16, 81-92.
[19] F. Brown, S. Narayan, R.S. Wahby, *et al.* (2017). Finding and Preventing Bugs in JavaScript Bindings. Proc of the IEEE Symposium on Security and Privacy (S&P'17), 559-578.
[20] G. Maisuradze, M. Backes, C. Rossow (2017). Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code. Proc of the 24th Annual Network and Distributed System Security Symposium, NDSS'2017.
[21] P. Oehlert (2005). Violating Assumptions with Fuzzing. IEEE Secur. Priv., 3(2), 58-62.
[22] P. Godefroid, M.Y. Levin, D.A. Molnar (2012). SAGE: Whitebox Fuzzing for Security Testing. Communications of the ACM, 55(3), 40-44.
[23] R. Swiecki, F. Gröbert. Honggfuzz. https://github.com/google/honggfuzz.
[24] M. Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.
[25] K. Serebryany (2016). Continuous Fuzzing with libFuzzer and AddressSanitizer. Proc of the IEEE Cybersecurity Development, SecDev'16, 157.
[26] M. Eddington. Peach Fuzzing Platform. http://community.peachfuzzer.com/WhatIsPeach.html.
[27] M. Sutton. Filefuzz. http://osdir.com/ml/security.securiteam/2005-09/msg0007.
[28] V.J.M. Manes, H. Han, C. Han, *et al.* (2019). The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering, 1.
[29] MozillaSecurity. JSfunfuzz. https://github.com/MozillaSecurity/funfuzz.
[30] J. Patra, M. Pradel (2016). Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data. Proc of the Tech. Rep. TUD-CS-2016-14664.
[31] J. Wang, B. Chen, L. Wei, Y. Liu (2017). Skyfire: Data-Driven Seed Generation for Fuzzing. Proc of the IEEE Symposium on Security and Privacy, S&P'17, 579-594.
[32] H. Han, D. Oh, S.K. Cha (2019). CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. Proc of the 26th Annual Network and Distributed System Security Symposium, NDSS'19.
[33] S. Lee, H. Han, S.K. Cha, *et al.* (2020). Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. Proc of the 29th USENIX Security Symposium, USENIX Security'20, 2613-2630.
[34] Ecma-TechnicalCommittee. Test262: ECMAScript Test Suite. https://github.com/tc39/test262.
[35] G. Ye, Z. Tang, S.H. Tan, *et al.* (2021). Automated Conformance Testing for JavaScript Engines Via Deep Compiler Fuzzing. Proc of the 42th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'21.
[36] M. Böhme, V. Pham, A. Roychoudhury (2016). Coverage-Based Greybox Fuzzing as Markov Chain. Proc of the ACM SIGSAC Conference on Computer and Communications Security, CCS'16, 1032-1043.
[37] M. Böhme, V. Pham, M. Nguyen, A. Roychoudhury (2017). Directed Greybox Fuzzing. Proc of the ACM SIGSAC Conference on Computer and Communications Security, CCS'17, 2329-2344.
[38] S. Gan, C. Zhang, X. Qin, *et al.* (2018). CollAFL: Path Sensitive Fuzzing. Proc of the IEEE Symposium on Security and Privacy, S&P'18, 679-696.
[39] J. Wang, B. Chen, L. Wei, Y. Liu (2019). Superion: Grammar-Aware Greybox Fuzzing. Proc of the 41st International Conference on Software Engineering, ICSE'19, 724-735.
[40] T.J. Parr, R.W. Quong (1995). ANTLR: A Predicated-LL(k) Parser Generator. Softw. Pract. Exp., 25(7), 789-810.
[41] H. Lin, J. Zhu, J. Peng, D. Zhu (2019). Deity: Finding Deep Rooted Bugs in JavaScript Engines. Proc of the 19th IEEE International Conference on Communication Technology, ICCT'19, 1585-1594.
[42] Y. Wang, Q. Wang, W. Ding (2020). Research on Semantic-Aware Fuzzing for JavaScript Engine. Journal of Information Engineering University, 21(03), 316-324.
[43] S. Groß(2018). Fuzzil: Coverage Guided Fuzzing for Javascript Engines, Department of Informatics, Karlsruhe Institute of Technology.
[44] C. Holler, K. Herzig, A. Zeller (2012). Fuzzing with Code Fragments. Proc of the 21th USENIX Security Symposium, USENIX Security'12, 445-458.
[45] T. Guo, P. Zhang, An, *et al.* (2013). GramFuzz: Fuzzing Testing of Web Browsers Based On Grammar Analysis and Structural Mutation. Proc of the International Conference on Informatics & Applications, ICIA'13, 212-215.
[46] S. Veggalam, S. Rawat, I. Haller, H. Bos (2016). IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. Proc of the 21st European Symposium on Research in Computer Security, ESORICS'16, 581-601.
[47] C. Aschermann, T. Frassetto, T. Holz, *et al.* (2019). NAUTILUS: Fishing for Deep Bugs with Grammars. Proc of the 26th Annual Network and Distributed System Security Symposium, NDSS'19.
[48] S. Park, W. Xu, I. Yun, *et al.* (2020). Fuzzing JavaScript Engines with Aspect-Preserving Mutation. Proc of the IEEE Symposium on Security and Privacy, S&P'20, 1629-1642.
[49] J. Park, S. An, D. Youn, *et al.* (2021). JEST: N+1 -Version Differential Testing of Both JavaScript Engines and Specification. Proc of the 43rd International Conference on Software Engineering, ICSE'21, 13-24.
[50] Google. ClusterFuzz: Scalable Fuzzing Infrastructure. https://google.github.io/clusterfuzz/.
[51] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://google.github.io/oss-fuzz/.
[52] O. Levi. Pin - a Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[53] DynamoRIO. Dynamic Instrumentation Tool Platform. https://dynamorio.org/.
[54] S. Schumilo, C. Aschermann, R. Gawlik, *et al.* (2017). KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. Proc of the 26th USENIX Security Symposium, USENIX Security'17, 167-182.
[55] D. She, K. Pei, D. Epstein, *et al.* (2019). NEUZZ: Efficient Fuzzing with Neural Program Smoothing. Proc of the IEEE Symposium on Security and Privacy, S&P'19, 803-817.
[56] Y. Wang, L. Sun, Y. Wang, Z. Xue (2021). A Fuzzing Method for JIT Complier of JavaScript Engine. Communications Technology, 54(01), 175-180.