# Survey on Javascript Engines Fuzzers

Lucas Bichet[1], Guillaume Guerard[1], and Soufian Ben Amor[2]

[1] Léonard de Vinci Pôle Universitaire, 92916 Paris La Défense, France
[2] LI-PARAD Laboratory EA 7432, Versailles University, 78035 Versailles, France
guillaume.guerard@devinci.fr

**Abstract.** The security of Web users is becoming a major issue today. Thus, Web browsers and more particularly the JavaScript engines that compose them are studied carefully to detect and promptly correct the vulnerabilities that they embed. This article presents the current state of research on vulnerabilities affecting JavaScript engines, starting from the current research context to the research tools used.

**Keywords:** Fuzzing · JavaScript Engine · Code Coverage.

## 1 Introduction

Since the end of the 1990s, the Internet has been spreading throughout the world at tremendous speed. It is the advent of the mass access, all the homes are then progressively equipped with an Internet connection. Software facilitating the navigation on the Web appears: the web browsers. Although these were very specialized at the beginning, they swiftly became extremely complex software, compatible with many protocols, specifications and interpreted languages.

These software are nowadays extremely used by both personal computers and phones, which makes them critical from the point of view of the security of users on the Internet. In particular, the JavaScript engines embedded in all web browsers nowadays are regularly subject to vulnerabilities discovered by researchers or exploited in the wild.

The notion of vulnerability (in the IT sense of the term) is well-known by the users. However, it is not as evident as it seems at first glance, and it is not always easy to explain what constitutes a vulnerability or not. A vulnerability can be described as an expected or unexpected feature that introduces a risk to the user's security, or to the target's continuity of operation.

Vulnerabilities have in common the fact that they have an impact on the security of users by definition. The method used to exploit the vulnerability and produce something interesting and concrete from the attacker's point of view is called "exploit" or "exploit code".

We will present the basics of the functioning of JavaScript engines to underline the research axes in term of security, then we will come back in more detail on the methods and tools accompanying the research of vulnerabilities on this subject.

Among the many software testing techniques available today, fuzzing has remained highly popular due to its conceptual simplicity, its low barrier to deployment and its vast amount of empirical evidence in discovering real-world software vulnerabilities [24]. Fuzzing refers to a process of repeatedly executing a program with generated inputs that may be syntactically or semantically malformed.

This paper focuses on the vulnerabilities through the JavaScript engine, how to detect the vulnerabilities. Following the literature review, we propose various axes of research to improve the vulnerabilities finding through fuzzer's enhancement.

The paper is built as follows: the section 2 exposes the notion of vulnerabilities and their consequences for a user. The section 3 presents the JavaScript language and engine. The section 4 shows the current research about how to discover vulnerabilities. The section 5 presents research axes to improve the researches. The section 6 concludes this paper.

## 2 Vulnerabilities

It is meaningful to distinguish bugs and vulnerabilities: a bug represents a programming error in computer code that makes it works in an unexpected way. Bugs can sometimes (but not necessarily) introduce vulnerabilities, while vulnerabilities can sometimes be the result of bugs but not necessarily either.

For example, a bug allowing a buffer stack overflow (writing outside the limits of a previously allocated buffer) probably introduces a vulnerability, while a bug causing an inability to read a file presumably does not.

In fact, the notion of vulnerability is subjective: it strongly depends on the context of the target as well as on the context of the target's users. For example, in the context of industrial activity monitoring software, the ability to remotely interrupt the software bears devastating consequences and therefore introduces an obvious security risk.

It is therefore legitimate to consider such an ability as a vulnerability. In the context of a Web browser, the ability to interrupt the software remains an annoying bug, but the security risks involved are limited: the notion of vulnerability is therefore probably unappropriate here.

### 2.1 Types of vulnerabilities

Since the notion of a vulnerability has been clarified through examples, we note typical characteristics that allow us to classify them by type such as:

1. Vulnerabilities known as injections [18, 34]: these are vulnerabilities induced by a misuse of inputs provided by a user. Typically, many Web vulnerabilities are injections that frequently come from the use of untrusted information provided by a user in an inappropriate context: SQL injections, JavaScript or XSS injections, etc.

2. Memory corruption vulnerabilities [5, 33]: these are low-level vulnerabilities occurring at the process memory management level. These vulnerabilities allow the arbitrary modification of the process memory by the attacker. For example, heap overflows are the best-known memory corruption, but many other vulnerabilities exist (use-after-free, type confusion or stack overflow [19]). These are the most common vulnerabilities found in JavaScript engines.
3. Logical vulnerabilities: these are vulnerabilities caused by a logical problem in the design of the application. These are habitually vulnerabilities on a larger scale (the entire information system). For example, an identification made through supposedly private data at one end of the information system, while this information can be gleaned publicly (and intentionally) elsewhere in the system.
4. Hardware vulnerabilities: these are vulnerabilities introduced by the hardware supporting a system or an application. This can be related to the design of the processor (Spectre, Meltdown vulnerabilities for example [30]), to the presence of an accessible console port offering elevated privileges on connected objects, etc.
5. Other types: vulnerabilities related to network protocols, access rights, etc.

### 2.2 Consequences of vulnerabilities

Vulnerabilities have in common the fact that they have an impact on the security of users by definition. The method used to exploit the vulnerability and produce something interesting and concrete from the attacker's point of view is called *exploit* or *exploit code*. In that manner, many ways to impact security can be grouped by:

1. Arbitrary code execution: the attacker obtains the ability to execute the code he requires on the targeted machine through the vulnerable application. Arbitrary code execution is one of the goals of an attacker targeting a JavaScript engine.
2. Exfiltration of private data: the attacker obtains the ability to gather data that he should not obtain access to (Cookies, passwords). The exfiltration of private data (like Cookies) can also be part of the goals sought by an attacker targeting a JavaScript engine.
3. Elevation of privileges: the attacker obtains the ability to perform privileged actions that he was unallowed to perform.
4. Denial of service: the attacker obtains the ability to alter the operation of the targeted vulnerable service. He can therefore interrupt it, slow it down, etc.

## 3 JavaScript

Let's focus on the JavaScript language and engine. JavaScript is an extremely widespread non-typed scripting language, initially used to make web pages dynamic. It has a significant need for performance, and this need tends web applications to rely on this language.

Because of its simple syntax, its asynchronous code mechanisms and its performance, JavaScript is equally spreading outside the Web and inviting itself in many Desktop applications via *Node.js* (derived from JavaScript) and *Electron.js* (a technology allowing porting a Web application in JavaScript in a Desktop application).

### 3.1 Javascript engines

A JavaScript engine is a software that interprets JavaScript code and allows its execution. JavaScript engines are part of the essential components of browsers with the rendering engines, the user interface, etc. There are several of them, used by different browsers:

1. JavaScriptCore[3] is used in Safari and developed by Apple. The JavaScript engine of Safari was also formerly called Nitro and SquirrelFish.
2. V8 is used in Google Chrome, Chromium (which itself represents the basis for many browsers such as Microsoft Edge, recent versions of Opera, Brave, Google Chrome) and Node.js in particular. It is developed by Google. Ross McIlroy, a well-known Google researcher, perfectly describes the V8 in the dedicated blog[4].
3. Chakra[5] is used in Internet Explorer 9 and developed by Microsoft.
4. SpiderMonkey[6] is used in Firefox and is developed by Mozilla.

The vast majority of JavaScript engines work in the same way and differ merely in their implementation (see Figure 1). These software are developed in C++ for the most part and are composed of two essential parts, the interpreter and the optimizer, and specific data structures (commonly called hidden classes, maps, shapes, etc.). Vulnerabilities can therefore be present at various levels of the JavaScript engine.

The information above is not intended to explain the functioning of a JavaScript engine but simply to highlight its key components, necessary for a proper understanding of the subject.

**Javascript interpreter** The JavaScript interpreter converts the JavaScript source code into an abstract syntax tree (AST) and generating the associated bytecode. This bytecode is then executed in a JavaScript virtual machine transcribing it on-the-fly into native machine code as shown in the Figure 2.

The interpreter is in charge of collecting information about the execution of the program such as the number of times a function is called, what type of arguments are passed to a function, etc. These data are stored in *caches* used by the JIT compiler during the optimization of the code.

[3] https://trac.webkit.org/wiki/JavaScriptCore
[4] https://v8.dev/
[5] https://github.com/chakra-core/ChakraCore
[6] https://www.mozilla.org/fr/js/spidermonkey/

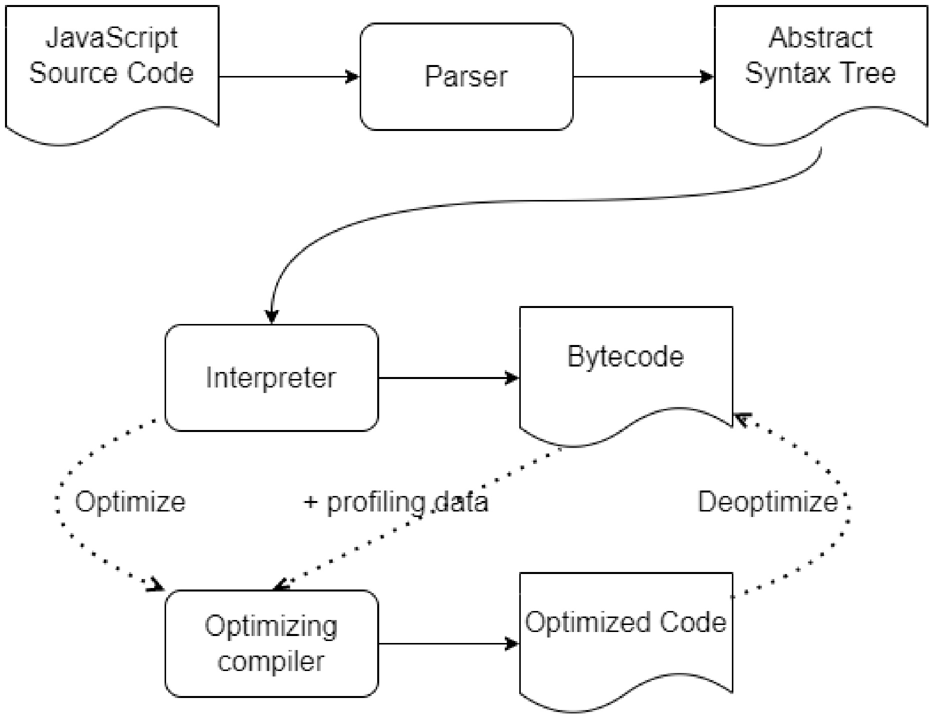
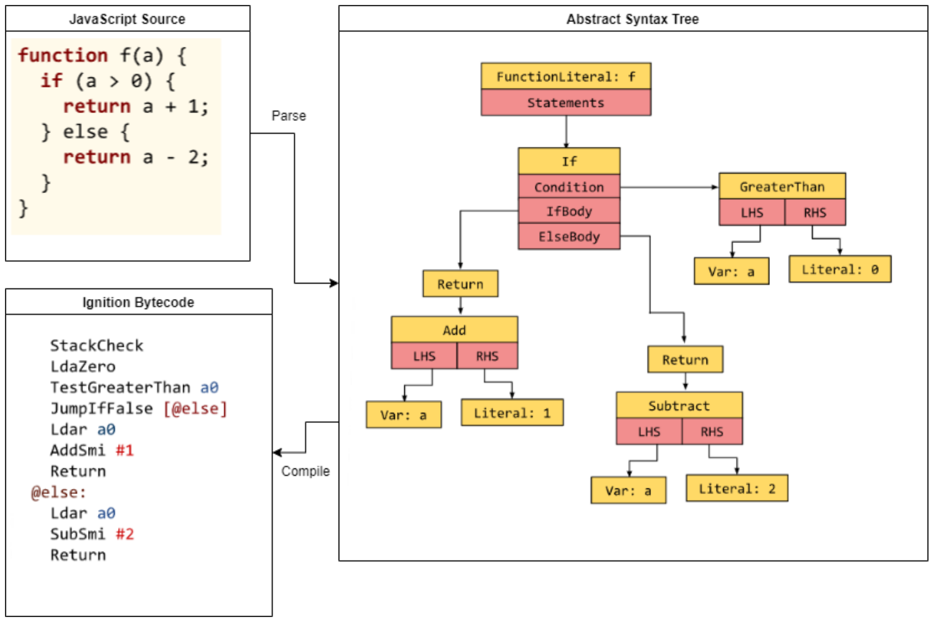


**Fig. 1.** General organization of a Javascript engine.

**Javascript Optimizer & JIT Compiler** The purpose of the JavaScript optimizer is to remove redundant parts of the code, and to translate the JavaScript bytecode of the functions very often executed into native machine code, which can be executed directly by the processor as presented in the Figure 3.

It is based on a compiler *Just-in-time (JIT)* or on-the-fly compiler that breaks down the optimization into several passes, which are specific to each JavaScript engine [35]. The information collected by the interpreter makes it possible to establish whether a function needs to be optimized or not: in the positive case, the JIT compiler converts the function into the form of a graph (as *sea of nodes* for V8 [29]) and proceed to the simplification of the graph of the function in several phases. For example, if it was noted that function A only received objects of a class B as arguments, function A will be specifically optimized for objects of class B. However, it is necessary to check that the arguments received by function A are always of type B to be able to deoptimize the function when its input arguments change and switch back to the non-specialized version of the function executed by the interpreter [31].

**Data structures** The principal difficulty of these engines is to succeed in recognizing the type of the data used, since JavaScript is untyped. For this, a data



```
function f(a) {
  if (a > 0) {
    return a + 1;
  } else {
    return a - 2;
  }
}
```

```
    StackCheck
    LdaZero
    TestGreaterThan a0
    JumpIfFalse [@else]
    Ldar a0
    AddSmi #1
    Return
  @else:
    Ldar a0
    SubSmi #2
    Return
```

**Fig. 2.** Generation of Javascript bytecode from source code.

structure is associated with the set of data used and contains all the information on the type of the data in question: its properties, its size, its accessors, etc. This data structure is found in all engines under different names: *Maps* in v8, *Shapes* in JavascriptCore, *HiddenClasses* more generally.

Many other optimizations are embedded in the code of some JavaScript engines, such as compression and pointer tagging for example.

## 3.2 Types of vulnerability

The most represented vulnerabilities found in Web browsers are memory corruptions like heap overflow [33] and integer overflow [15]. Some vulnerabilities are relatively specific to JavaScript based software, given their particular mode of operation. The confusion vulnerabilities [2] represent a very good example, since they only happen when a software is mistaken about the type of the data it manipulates, a situation that is not very frequent in usual codes but that happen in JavaScript engines given their specific framework.

The interpreter and the JIT mentioned above as well as their numerous subparts are likely to contain vulnerabilities. However, it appears that most of the vulnerabilities come from the JIT compiler, which is a much more complex piece of software than the interpreter. To be specific, type confusion frequently occurs as a result of errors in the optimization or deoptimization of executed functions.

However, the exploitation of JavaScript engine's vulnerabilities and related techniques will be uncovered in this document, as it is a separate and complex

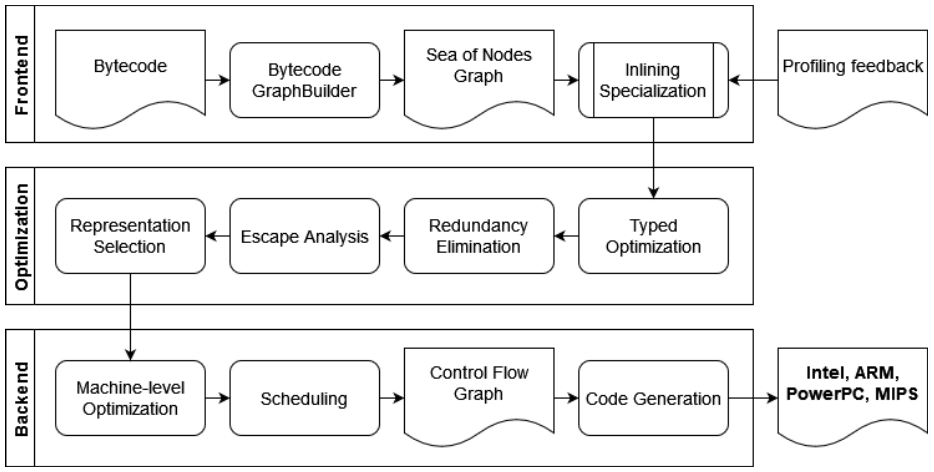


**Fig. 3.** Optimization of Javascript code (here with v8) ©JJY-security.

topic. There are known methods [13, 14, 32] allowing to convert a bug into an arbitrary code execution, most of them work as follows:

1. Triggering the vulnerability and manipulating the memory to corrupt a JavaScript object.
2. Obtaining primitives *addrof* and *fakeobj* allowing respectively to obtain the address in memory of a given JavaScript object and to construct an arbitrary fake JavaScript object in memory.
3. Obtaining arbitrary read and write primitives via the creation of a fake JavaScript array where its data pointer is modified thanks to the preceding primitives.
4. Execution of arbitrary code by the preceding primitives using various techniques to obtain RWX pages in memory (function compiled by the JIT or WebAssembly function).

JavaScript is a widespread language. Its specification is constantly evolving and JavaScript engines are constantly being equipped with recent and experimental features that are interesting areas of vulnerability research. One example is WebAssembly (WASM), which is a pseudo assembly language that offers an intermediate representation between JavaScript code and machine code while remaining independent of the underlying hardware. This feature is examined from a security point of view because it brings many new features to the code base and thus new horizons in terms of vulnerability research [16].

## 3.3 Vulnerability scanning techniques

There are essentially two principal ways to seek for vulnerabilities in web browsers and more generally in software.

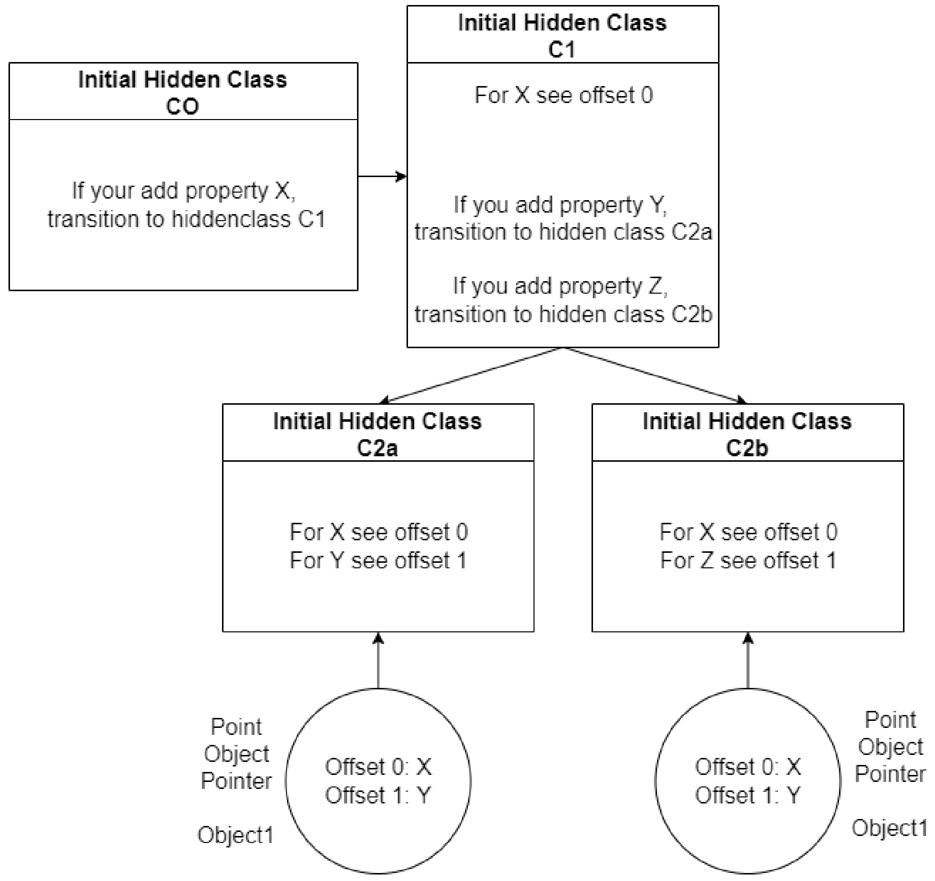


**Fig. 4.** Example of an HiddenClass containing information related to the type of the manipulated data.

Firstly, the manual code review consists in methodically reading the source code or the native code if the project is not open-source and trying to perceive errors. This technique is effective, but it is extremely time-consuming and tedious. However, it additionally allows for identifying undetectable details with the second technique, because a trained human eye will be skilled to assume the places where mistakes could have been made.

Secondly, the *fuzzing* consists of automatically generating inputs and submitting them to the target software. It is enough to monitor how the software behaves (crash, slowness, unusual behavior) to isolate inputs that probably lead to vulnerabilities. This technique is often very efficient but less accurate. It is equally necessary to configure a tool or even to develop one when striking a very specific target. The inputs are chosen or altered thanks to various techniques like genetic algorithms.

Fuzzing was introduced by Miller et al. [26] for evaluating the robustness of UNIX utilities against unexpected inputs. The difference between fuzzing and other black-box test generation methods is that fuzzing relies on very weak oracles-checking only for crashes or hangs, which lets it explore the input space automatically.

The optimization of the functioning of a fuzzer is a very vast domain: we can try improving the mutation algorithms generating the inputs, find new metrics to evaluate the impact of the tested input, improve the speed of the fuzzer, etc. We will therefore present in the following section the techniques and tools of fuzzing mostly used for the study of JavaScript engines.

## 4 Fuzzing of JavaScript engines

There are many well-known fuzzers whose efficiency has been proven. Some are intended to be non-specialized such as AFL++ [9], which is also very modular and therefore very reused. But in some cases, the target program taken as input data is structurally complex that a general fuzzer becomes insufficient and therefore useless. Indeed, when the generation of a valid input for the program is non-trivial, it becomes necessary to develop specially adapted fuzzing tools.

In the case of the study of a JavaScript engine, the generation of valid JavaScript code is essential if one wants to be able to fuzz in-depth the program, and in particular its JIT compiler. Indeed, if the JavaScript codes passed in the input are invalid, they will not even pass the stage of *parsing* and *interpretation*. Thus, it will never be executed and even less optimized by the JIT compiler. We will therefore only refer to the submerged part of the iceberg that is the parser integrated into the JavaScript engine.

We observe that all the solutions of fuzzing employed today to answer these constraints are more or less based on the use of the JavaScript grammar for the generation of code and the use of the code coverage as a metric to guide the fuzzing through mutation of the generated code or grammar [6, 24]. Although all JavaScript engine fuzzers are based in some way on grammar, their usage can vary depending on the situation.

Two classes of fuzzers have been developed, *generative* and *mutational* fuzzers. Generative approaches build a new test case from the ground following predefined rules like a context-free grammar of the JavaScript programming language or reassembling synthesizable code bricks dissected from the input corpus; mutational approaches synthesize a test case from existing seed inputs and adapt them for upcoming tests.

### 4.1 Fuzzing & feedback

In addition to the result of execution, it is conventional for fuzzers to utilize methods to obtain additional feedback after each execution to better guide the fuzzing.

We can separate fuzzers into several distinct categories based on the type of feedback they use:

1. The *white box fuzzers* recover during the execution the exact conditions to which the input is subjected. They modify the input in a suitable way to pass new conditions in the subsequent executions. The white box fuzzing is based on binary instrumentation techniques such as symbolic execution [11].
2. The *grey box fuzzers* only retrieve some information such as the code coverage: for each entry, they retrieve the code paths in the AST reached by this execution to be able to guide the next executions and encourage the exploration of new paths [37].
3. The *black box fuzzers* do not use any additional feedback. They use the target program as a black box and do not analyze how inputs to the program are processed. As a result, they are frequently faster than the first two but less efficient.

The vast majority of current fuzzers are grey-box fuzzers using code coverage as the primary metric. JavaScript fuzzers are no exception to this trend, and use code coverage as a clue to the interest of an entry for fuzzing.

### 4.2 News on Javascript's Fuzzers

We have chosen to focus on several fuzzers from different Javascript engines, in order to list and compare the notable features they deploy. These fuzzers have been chosen because they have produced good results, are recent and each introduces different novelties. There are many other interesting fuzzers [1, 7, 28, 36, 37], some of them will illustrate the next section about axes of research.

**CodeAlchemist** Presented in 2019, CodeAlchemist [17] exploits the notion of semantic validity in the generation of JavaScript code. Indeed, by relying on grammar or by applying mutations to pre-selected codes (called seeds), they guarantee the produced code will be syntactically valid i.e. it will be considered as valid code, but nothing guarantees its validity in a precise context. The sequence of lines of code may not produce any sense, and this is precisely what CodeAlchemist intends to prevent. It extracts code bricks from the seeds provided as input, and associates constraints to them as rules to follow when connecting blocks together. The fuzzer is thus separated into three parts: the seeds parser, the constraints analyzer, and the fuzzer itself.

**Deity** Introduced in 2019, Deity [23] employs the notion of fuzzing guided by abstract syntax tree mutation (AST). In fact, Deity manages a abundant number of seeds for its mutation algorithm. These seeds come from JavaScript code recovered on the Internet but also from newly discovered vulnerabilities since we can observe they are often triggered by the same exploits. The idea is to convert these JavaScript codes into AST as an interpreter would do and to operate mutations directly on this AST: deletion of nodes, merging of several paths coming from various codes. Once the mutation has been applied, the algorithm redoes the reverse conversion to recover valid and mutated JavaScript source codes, which are submitted to the targeted engine.

**EvoGFuzz** Presented in 2019, EvoGFuzz [8] has the particularity to apply mutations not on codes generated by a grammar, but directly on the grammar itself. It requires seeds and a file describing the JavaScript grammar. At the start, it will generate probabilistic grammar by isolating the most used grammar rules in the seeds. Then, a population of codes to submit to the JavaScript engine is generated, and a genetic algorithm is used to select the most interesting entries. The structure of these entries will then influence the defined grammar, and thus the next generation of entries.

**FuzzILi** Presented in 2018, FuzzILi [12] is a fuzzer also employing a mutative approach. It has the particularity to use an intermediate representation language specially created for the occasion: FuzzIL. This language takes the form of a both easily mutable and easily convertible bytecode into JavaScript code. Then, they exert several kinds of mutations to it: *input mutator* changes the variables used by the instructions, *combine mutator* combines two already existing intermediate representation languages. To conclude, the code coverage provides information about the interest of the newly created mutation.

**Montage** Presented in 2020, Montage [21] starts with two interesting observations:

- Vulnerabilities often come from files previously patched against other vulnerabilities
- Code fragments that trigger vulnerabilities frequently reuse code snippets from existing test code.

Based on these observations, Montage uses a Neural Network Language Model (NNLM) to learn the links between the code fragments found in the test sets. The fuzzing is done in three phases:

- Firstly, it consists of extracting code fragments from the existing test sets.
- Then, the model trains the NNLM on the basis of the extracted fragments.
- Finally, the model generates newly potentially vulnerable codes thanks to the NNLM previously trained.

## 5 Research axes

Fuzzing is the most popular approach for discovering vulnerabilities in JavaScript engines. Many fuzzers exist and have already been proven, but most of the time they are only made public when they have already been overexploited on known open-source JavaScript engines. It is therefore possible to draw inspiration from them, to reuse them on other open-source engines, but the exploration of creative ideas and tracks remains essential to discover more vulnerabilities. We identify five potential axes of improvement.

## 5.1 Hybrid methods

Hybridization consists in designing a new tool by combining several projects and in particular the novelties introduced by the different fuzzers presented. The idea is to benefit from the advances of fuzzers, to complete others, and thus obtain better efficiency in the fuzzing.

Outside JavaScript fuzzing, Yun et Al. propose QSYM [39], a hybrid fuzzer for real-world programs' binaries, which uses Dynamic Binary Translation (DBT) to natively execute the input binary as well as to select basic blocks for symbolic execution. The DBT produces basic blocks for native execution and prunes them for symbolic execution, allowing to switch between two execution models. Then, QSYM selectively emulates only the instructions necessary to generate symbolic constraints. The fuzzer is used on the LAVA-M dataset and outperformed bug-finding tools like Driller and VUzzer.

Zhao et Al. observe hybrid fuzzing which combines fuzzing and concolic execution has become an innovative technique for software vulnerability detection (especially on binaries). They propose DigFuzz [40]. They design a novel Monte Carlo-based probabilistic path prioritization model to quantify each path's difficulty and prioritize them for concolic execution. They test their fuzzer on the LAVA dataset and outperformed Driller and AFL.

Kim et Al. work on Linux kernel [20]. They propose a hybrid fuzzing with three features: 1) converting indirect control transfers to direct transfers, 2) inferring system call sequence to build a consistent system state, and 3) identifying nested arguments types of system calls. The proposed HFL fuzzer outperformed Moonshine and Syzkaller, the main fuzzers on Linux kernel.

To conclude about hybrid methods, the goal of those methods is to infer a program model or input grammar from either observing the behavior of the program on multiple inputs, using formal approaches, machine learning based on a previously available corpus, or observing and summarizing the program execution.

## 5.2 Algorithm improvements

Most of the fuzzers have drawbacks. Lexical approaches such as traditional fuzzing fail because of the sheer improbability to generate valid inputs and keywords, whereas the symbolic constraint solving of semantic approaches fail due to the combinatorial explosion of paths. Therefore, researchers aim to improve algorithms to avoid invalid inputs or paths.

Improving the algorithms consists of changing the metrics and the different algorithms used in the different parts of the fuzzer:

- Genetic algorithms for choosing which entries to keep according to their code coverage
- Input mutation algorithms
- Seed selection algorithms
- Etc.

Considering JavaScript engine, Park et Al. [28] propose a new fuzzer DIE, including a new technique called an aspect-preserving mutation, that stochastically preserves beneficial properties and conditions of the original seed input in generating a new test case.

Mathis et Al. present parser-directed fuzzing pFuzzer [25] as it specifically targets syntactic processing of inputs via input parsers by a dynamic tainting of input characters. It is able to generate syntactically valid inputs for a large class of programs, avoiding large input errors. Padhye et Al. propose Zest [27], which converts random-input generators into deterministic parametric generators. est leverages program feedback in the form of code coverage and input validity to perform feedback-directed parameter search.

Liang et Al. expose DeepFuzzer [22], an enhanced grey box fuzzer with qualified seed generation, balanced seed selection, and hybrid seed mutation. They use a symbolic execution approach to generate qualified initial seeds which then guide the fuzzer through complex checks. They apply random and restricted mutation strategies which are combined to maintain a dynamic balance between global exploration and deep search.

### 5.3 Improving directed fuzzing

JavaScript is a language in constant evolution (same for any back-end JavaScript runtime environment like *Node.js*), it is necessary to integrate the recent constructions and functions proposed by the standard such as "Array.prototype. reduce()" and the generating expressions in JavaScript 1.8. This can be performed by integrating adapted seeds, but also by adapting the mutation algorithms to encourage the generation of these constructs. Moreover, the control structures ("for", "if" ...) are globally little used by fuzzers, so it could also be interesting to strengthen the mutation algorithms so that they use more of these structures.

New code sources additionally include the recent features related to the code base of each JavaScript engine. By following the most recent additions to the code base, we see that new features in terms of optimization are gradually being introduced, which probably heralds the discovery of new vulnerabilities in these new areas. For example, V8 is introducing modern compilers to its optimization chain: Turboprop and Sparkplug[7], which are described as "midtier" compilers, capable of generating less optimized code than the current JIT compiler, but much faster.

The process of adding new features in fuzzer is mostly done by generating a new grammar for the new version. When considering seeds that suit the tested program, we called the process directed fuzzing. Established techniques such as pattern recognition, inference, and feedback have been developed. Researchers tend to enhance directed fuzzing by including deep-learning methods to discover alternative directions to fuzz.

Böttinger et Al. [3] add a Q-Learning function (from deep learning theory) to reward some mutations. This process guides the fuzzer into inputs that better fit

[7] https://v8.dev/

the version or the program tested. Zong et Al. present FuzzGuard [41], a deep-learning-based approach to predict the reachability of inputs since nine out of ten inputs don't strike a bug in undirected fuzzing. FuzzGuard includes a 3-layer Convolutional Neural Network model to assume the impact of inputs before the test process, avoiding making inputs that will not return results.

## 5.4 Code coverage of optimized functions

Practically, tracking full and accurate path coverage is infeasible in practice due to the high instrumentation overhead. Thus, the fuzzers include algorithms to cover the code.

Wang et Al. propose SAFL [38], including a coverage-directed mutation. This fair and fast algorithm helps the fuzzing process to exercise rare and deep paths with more significant probability. Gan et Al. present CollAFL [10], a coverage-sensitive fuzzing solution. It mitigates path collisions by providing more accurate coverage information, while still preserving low instrumentation overhead. It also utilizes the coverage information to execute three innovative fuzzing strategies, promoting the speed of discovering alternative paths and vulnerabilities.

Code coverage is currently achieved by including the necessary code to the JavaScript engine source code before it is compiled. The compiled code generated during the initial compilation is correctly covered and instrumented, but this is not the case for the compiled code generated by the JIT compiler during the execution of the engine. It would be extremely interesting to be able to obtain information about the code coverage in these on-the-fly compiled functions, we could specifically fuzz them by varying the inputs to them according to the code coverage feedback.

## 5.5 Generic fuzzing

Most of the current fuzzers require the source code of the engine they are interested in, mainly to include to it the parts of code necessary to obtain information like code coverage. This requirement remains not really a problem since all mainstream browsers use open-source JavaScript engines. However, it might be interesting to consider closed-source JavaScript engines.

It would probably be necessary to find a way to cover the code that does not require the source code. AFL++ is a generic fuzzer project proposing to implement the emulation of an executable via QEMU to provide this kind of information [9]. Chen et Al. propose PolyGlot [4] a fuzzing framework that can generate semantically valid test cases to extensively test processors of different programming languages. They design a uniform intermediate representation to neutralize the difference in the syntax and semantics of programming languages in the Backus-Naur form.

## 6 Conclusion

Security vulnerabilities in software may lead to serious consequences, and vulnerability exploitation has become a hot area of research in networks and information security. Along with the expansion of complexity of software and JavaScript engines, fuzzing has incomparable advantages which other vulnerability exploiting technology can't provide such as static analysis.

The literature on fuzzing, and more precisely on JavaScript's fuzzers, is blooming since 2018 with various new techniques. The genetic evolution theory and the development of new algorithms for seeds selection and trees pathfinding and merging have greatly improved the ability and accuracy of fuzzers. We provide in this paper a review of recent trends in fuzzing on JavaScript engines like model inference, hybrid fuzzing, and new genetic algorithms. Furthermore, we provide several axes to improve the fuzzing algorithms based on some recent and promising works. We also provide unexplored and specific axes to the JavaScript engines.

## References

1. Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.R., Teuchert, D.: Nautilus: Fishing for deep bugs with grammars. In: NDSS (2019)
2. Bishop, M.: Vulnerabilities analysis. In: Proceedings of the Recent Advances in intrusion Detection. pp. 125–136 (1999)
3. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 116–122. IEEE (2018)
4. Chen, Y., Zhong, R., Hu, H., Zhang, H., Yang, Y., Wu, D., Lee, W.: One engine to fuzz'em all: Generic language processor testing with semantic validation. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P 2021) (2021)
5. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th international conference on World wide web. pp. 281–290 (2010)
6. Dewey, K., Roesch, J., Hardekopf, B.: Language fuzzing using constraint logic programming. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 725–730 (2014)
7. Dominiak, M., Rauner, W.: Efficient approach to fuzzing interpreters. BlackHat Asia (2019)
8. Eberlein, M., Noller, Y., Vogel, T., Grunske, L.: Evolutionary grammar-based fuzzing. In: International Symposium on Search Based Software Engineering. pp. 105–120. Springer (2020)
9. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++: Combining incremental steps of fuzzing research. In: 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20) (2020)
10. Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z.: Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 679–696. IEEE (2018)
11. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 206–215 (2008)

12. Groß, S.: FuzzIL: Coverage guided fuzzing for JavaScript engines. Ph.D. thesis, Master's thesis, Karlsruhe Institute of Technology, 2018. https://saelo . . . (2018)
13. Groß, S.S.: Attacking javascript engines : A case study of javascriptcore and cve-2016-4622 (2016), http://www.phrack.org/papers/
14. Groß, S.S.: Jit exploitation (2019), http://www.phrack.org/papers/
15. Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for javascript. ACM SIGPLAN Notices **47**(6), 239–250 (2012)
16. Hamidy, G., et al.: Differential fuzzing the webassembly (2020)
17. Han, H., Oh, D., Cha, S.K.: Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In: NDSS (2019)
18. Johari, R., Sharma, P.: A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection. In: 2012 International Conference on Communication Systems and Network Technologies. pp. 453–458. IEEE (2012)
19. Kang, Z.: A review on javascript engine vulnerability mining. In: Journal of Physics: Conference Series. vol. 1744, p. 042197. IOP Publishing (2021)
20. Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B.: Hfl: Hybrid fuzzing on the linux kernel. In: Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA (2020)
21. Lee, S., Han, H., Cha, S.K., Son, S.: Montage: A neural network language model-guided javascript engine fuzzer. arXiv preprint arXiv:2001.04107 (2020)
22. Liang, J., Jiang, Y., Wang, M., Jiao, X., Chen, Y., Song, H., Choo, K.K.R.: Deep-fuzzer: Accelerated deep greybox fuzzing. IEEE Transactions on Dependable and Secure Computing (2019)
23. Lin, H., Zhu, J., Peng, J., Zhu, D.: Deity: Finding deep rooted bugs in javascript engines. In: 2019 IEEE 19th International Conference on Communication Technology (ICCT). pp. 1585–1594. IEEE (2019)
24. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering (2019)
25. Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., Zeller, A.: Parser-directed fuzzing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 548–560 (2019)
26. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM **33**(12), 32–44 (1990)
27. Padhye, R., Lemieux, C., Sen, K., Papadakis, M., Le Traon, Y.: Semantic fuzzing with zest. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 329–340 (2019)
28. Park, S., Xu, W., Yun, I., Jang, D., Kim, T.: Fuzzing javascript engines with aspect-preserving mutation. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1629–1642. IEEE (2020)
29. Report, F.I.: Sea of nodes in v8 (2015), https://darksi.de/d.sea-of-nodes/
30. Schwarz, M., Lackner, F., Gruss, D.: Javascript template attacks: Automatically inferring host information for targeted exploits. In: NDSS (2019)
31. Sevcik, J.: Deoptimization in v8 (2016)
32. @sirdarkcat, t.: Eat sleep pwn repeat browser training (2019)
33. Sotirov, A.: Heap feng shui in javascript. Black Hat Europe **2007**, 11–20 (2007)
34. Thiyab, R.M., Ali, M., Basil, F., et al.: The impact of sql injection attacks on the security of databases. In: Proceedings of the 6th International Conference of Computing & Informatics. pp. 323–331 (2017)
35. Titzer, B.L.: Turbofan jit design

36. Wang, J., Chen, B., Wei, L., Liu, Y.: Skyfire: Data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 579–594. IEEE (2017)
37. Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: Grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 724–735. IEEE (2019)
38. Wang, M., Liang, J., Chen, Y., Jiang, Y., Jiao, X., Liu, H., Zhao, X., Sun, J.: Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 61–64 (2018)
39. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 745–761 (2018)
40. Zhao, L., Duan, Y., Yin, H., Xuan, J.: Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: NDSS (2019)
41. Zong, P., Lv, T., Wang, D., Deng, Z., Liang, R., Chen, K.: Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp. 2255–2269 (2020)