Concolic-Fuzzing of JavaScript Programs using GraalVM and Truffle

Robert Delhougne[1]

Abstract: The scripting language JavaScript has established itself as a central component of the modern internet. However, the dynamic execution model of the language limits the support for source-code analysis, which leaves a developer without essential tools to maintain safety and security requirements. This paper describes a concolic-fuzzer based on the GraalVM to automatically test JavaScript programs. The fuzzer shows promising results in both code coverage and runtime evaluations and provides developers with additional features such as special analysis targets.

Keywords: Software-Verification; Fuzzing; Concolic-Fuzzing; JavaScript; GraalVM; Truffle

1 Introduction

Originally intended as a simple language for interactive media on a web application's client-side, the scripting language JavaScript is now also used for server- and desktop applications. However, a study by Ocariza et al. [OPZ11] shows that programming with JavaScript can be difficult because even subtle programming errors can lead to severe repercussions later in the program's execution. Moreover, due to the aggressive type-coercing, these errors can propagate through the program for a long time before being detected and thus hiding the initial cause [PS15]. These features of the language and their consequences complicate the development process and point to the need for more precise tools for checking code quality.

In this paper, the author presents a tool that can automatically test JavaScript programs for errors with a technique called *concolic-fuzzing*, an advanced software-testing method that has gotten more attention in recent years [Ba18; Ka15]. This fuzzer uses the virtual machine GraalVM and its language implementation framework Truffle to execute the JavaScript programs and keep track of the inner workings of the program to guide the testing process. One advantage of this method is that it is fully automated and does not require any modification of the language or program to guide the testing process. To address the difficulties of JavaScript, the fuzzer has special analysis targets to detect the errors characterized by Ocariza et al. [OPZ11] or Pradel and Sen [PS15] as early as possible. In a preliminary evaluation, the fuzzer reached between 50% to 98% branch coverage without the need to specify any information about the input structure of the test programs.

[1] TU Dortmund, August-Schmidt-Straße 4, 44227 Dortmund, Germany, robert.delhougne@tu-dortmund.de

Related Work One of the first tools using concolic-execution is DART, short for *directed automated random testing* [GKS05]. DART can be used to test the programming interfaces of C-programs automatically. Build on the ideas of DART, the testing framework jDART proposed by Luckow et al. [Lu16] can test Java programs. The concolic-execution of the program is performed using the Java-Pathfinder [Vi03] framework. Another well-known program for symbolic execution is SAGE [GLM08]. Godefroid et al. are stating that it was extensively used in the development of the operating system Windows 7, claiming that about one-third of all errors found with testing based on files are found by this tool. SAGE works by directly utilizing x86 instructions and thus is capable of testing all programming languages that can be compiled to x86 machine-code. The support for dynamic languages like JavaScript in symbolic execution tools is much less common. Saxena et al. [Sa10] presented a symbolic execution framework for JavaScript called *Kudzu*. Kudzu uses a simplified version of the JavaScript language to reduce the complexity of the symbolic model called *JASIL*. The tool is used to test inputs for web applications and utilizes a modified version of the WebKit engine to observe the application at runtime. Sun et al. [Su18] also presented a software testing framework for Node.js applications that utilizes source-code instrumentation, but does not use true symbolic or concolic execution techniques like they are employed in this work.

2 Motivational Example

```
function factorial(n) {
  if (n >= 0) {
    fac = 1;
    while (n > 1) {
      fac = fac * n;
      n = n - 1;
    }
    return fac;
  }
}
```

List. 1: Algorithm to calculate the factorial of n.

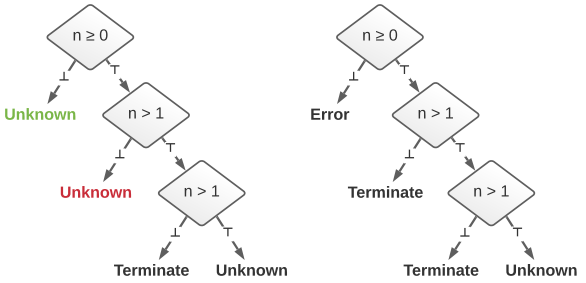


Fig. 1: Execution trees of `factorial(n)`.

To visualize the methodology of fuzzing in general and specifically the principles of concolic-fuzzing, we look at the small JavaScript function shown in Listing 1. The function call of `factorial(n)` with $n < 0$ introduces an error state into the surrounding program (e. g. `factorial(-1)` returns the value `undefined`). To test this function, besides approaches like unit testing, the automated technique *fuzzing* can be used. This method aims to generate a vast number of (pseudo-) random input data to test the software. If the software reaches a failure state, the fuzzer detects it and saves the corresponding input data. However, a naive, completely random fuzzer often has a low probability of triggering errors in a program

and needs redundant test iterations. Modern fuzzers use additional information about the inputs or the program to narrow down the generated input data and speed up the testing circumventing this issue. One of these advanced techniques used in this paper is called *concolic-fuzzing*.

Let's demonstrate concolic-fuzzing using the example program in Listing 1. In the first step, the fuzzer executes the program with random inputs, in this example the input of function `factorial(n)` is set to $n = 2$. Simultaneously with running the program with this concrete value, the fuzzer treats the input as a *symbolic* value and records all modifications to this variable. This implies that the fuzzer can fully observe all procedures of the tested program at runtime, down to single operators and variable accesses. With the help of the symbolic variables, the fuzzer constructs an execution tree to keep track of the already executed paths of the program. Figure 1 (left) shows the execution tree for function `factorial(n)` after the first iteration with $n = 2$. For every new test iteration, the fuzzer chooses a different, unknown execution path from this model, e. g., the red "Unknown" node. All the branch conditions starting from the root of the tree down to the unknown location are then collected and combined to form a *path-condition*:

$$n \geq 0 \quad \wedge \quad \neg(n > 1). \tag{1}$$

To find a value for n that fulfills these conditions, the fuzzer uses an *SMT-Solver* like the solver Z3[2] developed by Microsoft. In this example, the only possible solution is $n = 1$. If the program is executed with this value as an input, it takes exactly[3] this predicted execution path, and the fuzzer can extend the execution tree. To trigger the failure state in the program, the fuzzer only has to collect the conditions for the green node (only $\neg(n \geq 0)$) and then run the program with an n satisfying this condition. Figure 1 (right) also depicts the execution tree after these two iterations.

With this method, a concolic-fuzzer quickly achieves very high code coverage because all new inputs are tailored for a new execution path. Code paths that check the input data for distinct patterns and usually pose a hard to overcome barrier for naive fuzzers can easily get around.

3 Implementation

As demonstrated in the example, a concolic-fuzzer needs to observe the tested program at runtime and must be able to modify the execution e. g. to inject new input values directly into the running program instead of relying on reading the data from the file system. In this implementation, I use the GraalVM and the Truffle framework for this purpose. The

[2] `https://github.com/Z3Prover/z3`

[3] There are exceptions, e. g., if the program flow depends on a non-deterministic variable or the fuzzer does not exhaustively model the program flow.

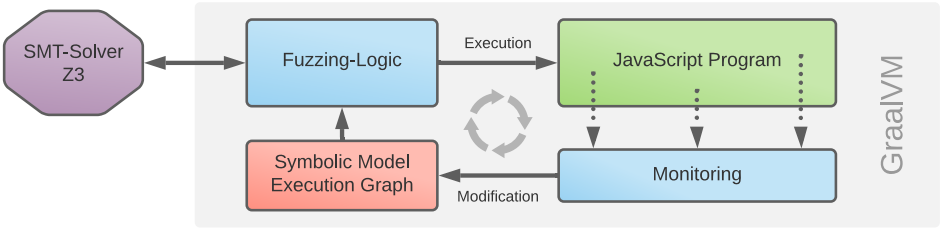


Fig. 2: Overview of the fuzzer's architecture.

GraalVM is a virtual machine for Java, developed by Oracle, based on the HotSpot VM. Its first production-ready version was released in 2019. Besides Java, it supports a wide variety of additional languages through Truffle, which is a framework to implement an interpreter for *abstract-syntax-trees* (ASTs) [Wü13]. The behavior of the interpreted language, the so-called *guest-language*, is implemented through Java classes. One benefit of the Truffle architecture is that it also allows developers to deeply inspect and alter the execution of a guest-language program at runtime, based on the AST representation of the program. In the concept of GraalVM, these extensions are called *tools*.

Figure 2 shows the simplified architecture of the fuzzer. All of the components, except the SMT-Solver, are running inside of the GraalVM. The program logic of the fuzzer creates a *language-context*, that parses the source code of the executed program, builds the AST, and then executes it. Simultaneously, the fuzzer monitors all operations of the AST and creates the symbolic model of the program (cf. Section 3.1). The execution tree grows with every iteration and based on that, the fuzzer chooses a new execution path according to a specific strategy (cf. Section 3.2), solves the path constraints with the help of an SMT-Solver (in this case, Z3), and executes the JavaScript program again with the newfound input values. When the program is running, the fuzzer checks it for exceptions or special error analysis targets (cf. Section 3.3). The three main components of the implementation are presented in more detail in the following subsections.

3.1 Symbolic Flow

To keep track of the variable modifications inside the running JavaScript program, the fuzzer uses the Truffle API concept of *wrapper-nodes*. In this work these nodes are used to define a symbolic behavior for every AST-node of the program, in addition to the concrete behavior that is already given by the language implementation. The symbolic behavior depends on the type of the AST-node, for example, arithmetic nodes, variable read/write or constants. The wrapper-nodes are dynamically attached to nodes in the (JavaScript-) AST and can listen to specific *events*, e. g., before the corresponding AST-node gets executed, a new input is available, or execution of the node has finished. With the help of the wrapper nodes, a developer can extend the behavior of the AST-nodes as desired. The symbolic model of the

program, modified by the wrapper-nodes, consists of two central data structures that are essentially representing a load-store-architecture.

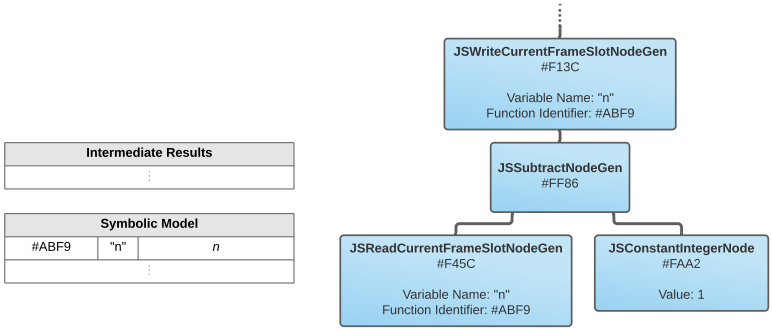
Intermediate Results The fuzzer saves all symbolic counterparts to intermediate results, computed by the AST-nodes, into a data structure consisting of a unary function $i \rightarrow S$, mapping integer identifiers i onto a data structure for symbolic expressions S. The integer i is a unique identifier of an AST-node of the instrumented JavaScript program. This can be thought of as symbolic "register".

Symbolic Model The symbolic model is a binary function $k \times s \rightarrow S$, which contains a flat hierarchy of the symbolic representation of all the data structures currently present in the observed JavaScript program. The integer identifier k, in this case, represents a JavaScript object, the string s represents the attribute name. This data structure can model attributes inside objects, arrays, and stack frames. This data structure is a symbolic "memory", where the intermediate results get written to when their concrete counterpart is assigned to a JavaScript variable.

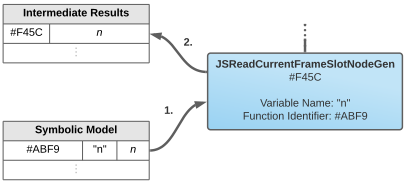
Figures 3a to 3e show an example of the technique for the JavaScript expression `n = n - 1` that is part of the small JavaScript function `factorial(n)`, shown in Listing 1. The expression reads a local variable n, subtracts 1 and saves the result back to the local variable. The initial situation is shown in Figure 3a. First, given the information about the current function context, the `JSReadCurrentFrameSlotNodeGen` loads the symbolic expression n from the variable "n" of the current function from the symbolic model into the intermediate results (Figure 3b). Then, the `JSConstantIntegerNode` is evaluated (Figure 3c). The wrapper of this node only stores a symbolic constant into the intermediate results. Both of these intermediate results are then used by the instrumentation of the `JSSubtractNodeGen` to construct the symbolic subtraction operation (Figure 3d). In the last step, this symbolic result is transferred back to the symbolic model, just like it is saved as a local variable in the simultaneously running concrete execution of the program (Figure 3e).

3.2 Path Exploration

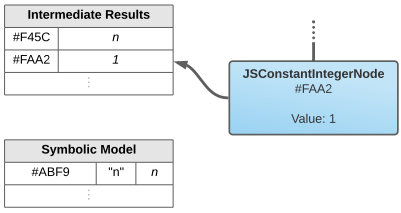
To construct the execution tree of the program, two types of AST-nodes have to be instrumented: The node type `IfNode` for branching instructions and the node type `WhileNode` to cover `while` and `for` loop statements. A separate node type handling `for`-loops does not exist; this loop statement is also handled by the `WhileNode`. The execution tree in this fuzzer is modeled as a state-machine, meaning every new iteration of the JavaScript program, the current position inside the execution tree is reset to the root of the tree. While the fuzzer runs the JavaScript program, the instrumentation of the `IfNodes` and `WhileNodes` keep track of the current position inside the tree. When the program enters an unknown part



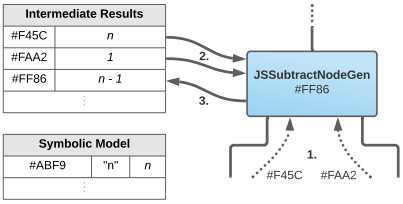
(a) Initial condition for the expression `n = n - 1` from the function `factorial(n)`.



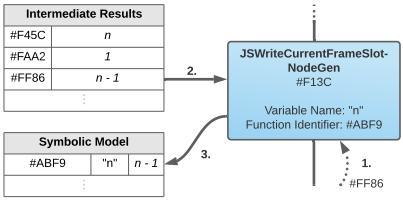
(b) Transfer of the symbolic value *n* from the model to the intermediate results.



(c) Creation of a new symbolic constant.



(d) Calculation of the symbolic subtraction operation.



(e) Transfer of the intermediate result back to the model.

Fig. 3: Example of the concolic execution based on AST-nodes.

of the tree, meaning this execution path is run for the first time, the tree gets extended by the instrumentation. The wrapper-node transfers the matching branch condition from the intermediate results to the newly created execution tree node.

One of the challenges of symbolic execution is the selection of the next possible execution path. In this work, I implemented three different strategies to select a new, unknown execution path. They are listed in Table 1. For the comparison of the strategies, refer to Section 4. The user can further configure these strategies and other parts of the fuzzer with

Strategy	Description
`DEPTH_SEARCH`	Of all the unknown candidate leaves of the execution tree, select the one that is the farthest away from the root of the tree.
`IN_ORDER_SEARCH`	Traverse the tree in-order and return the first unknown node that is found.
`RANDOM_SEARCH`	Traverse the tree from the root. At each node, switch into one of the child nodes with a possibility of $\frac{1}{2}$ until an unknown node is found. If a leaf is reached but it is not unknown, use backtracking.

Tab. 1: List of the implemented search strategies for unknown execution paths in the execution tree.

a YAML configuration file. One example of a parameter is the maximum search depth in the execution tree, which prevents the fuzzer from forming too long path predicates that are hard to solve in later stages of the test process.

3.3 Special Analysis Targets

To further adapt the fuzzer to the JavaScript language, I extended the fuzzer to analyze the tested program for common programming mistakes. A known problem of JavaScript, which is also described by Pradel and Sen [PS15], is that the aggressive type-coercing of JavaScript can suppress errors, for example, in the form of an undefined value, for a long time, before they are finally causing an unrecoverable error that is noticeable by the developer or user. This hides the actual cause of the error. To circumvent this problem, this work extends the fuzzer by enhanced error guards. These guards describe a valid JavaScript behavior under normal circumstances that the fuzzer treats as an error condition when the user enables the error guard during the test process. An example is the guard `division_op_no_zero` that prevents the evaluation of a zero-division to `+Infinity` and instead throws an exception that the fuzzer handles as a faulty state of the program. With the corresponding input value and line number, the developer can reproduce the error at the root cause. These extensions to the expected behavior of JavaScript applications also testify to the power and versatility of the code instrumentation with the Truffle API.

4 Results

The evaluation objectives for the project are the the effectiveness of the three implemented search strategies for path exploration and the total runtime of the testing process. The evaluation set consists of about 15 JavaScript programs, ranging from simple functions to complex programs with hundreds or thousands of lines. Table 2 shows a selection of these programs and their properties. The programs are from three different sources: First, programs written by the author, which are small with few inputs. Secondly, the work used programs from the “Benchmarksgame”[4] project. Third, the evaluation uses several

[4] `https://benchmarksgame-team.pages.debian.net/benchmarksgame`

Name	Symbolic Inputs	LOC	Max. cycl. Compl.	# Func.
`fasta.js`	1 String	106	6	4
`calculator.js`	2 Integer, 1 String	296	23	22
`infusion.js`	37 Integer, 22 Boolean	1165	70	23
`alarm.js`	65 Integer, 43 Boolean	2143	66	28
`minepump.js`	2 Integer, 16 Boolean	331	19	38
`addition01.js`	2 Integer	36	6	2
`triangle.js`	3 Integer	46	8	2

Tab. 2: A selection of test programs and their properties. Listed are the number and types of the symbolic input variables for each program, the number of lines (LOC), the maximum cyclomatic complexity per function, and the total number of functions.

programs from the SV-COMP[5] benchmark, which have been converted manually from Java to JavaScript. All of the measured coverage metrics and runtimes were calculated on a test system, consisting of an AMD Ryzen 7 4800HS (16 vCores) and 16 GiB of RAM. For detailed information about the used software versions please refer to the project page[6] on GitHub.

4.1 Code Coverage Metrics

To evaluate the effectiveness of the fuzzing technique in general and compare the different path exploration strategies, the fuzzer calculates three different coverage metrics: statement coverage (C_0), function coverage, and branch coverage (C_1). This evaluation will focus on branch coverage. Figure 4 shows the calculated metric for some of the example programs. I configured the fuzzer to stop the testing on a maximum of 3000 iterations or if it reaches 95% branch coverage. Moreover, I configured the search strategies to search in a maximum depth of 64 in the execution tree. As the `RANDOM_SEARCH` strategy is nondeterministic , I performed three different runs for this strategy to visualize the differences per run. In the first execution of the test programs, the fuzzer always executes the programs with default values for all inputs, namely `true` for boolean inputs, `1` for numeric values, and the empty string for string inputs.

While the fuzzer quickly improves the code coverage against the first run in some of the programs, namely `calculator.js`, `infusion.js`, or `triangle.js`, other programs like `alarm.js` or `minepump.js` are less suited for this testing technique. This is likely since these programs are simulating state machines. These programs prevent the execution of certain code paths due to their structure, so a coverage like in `alarm.js` cannot be improved much, even with many iterations. The fuzzer improves the test coverage of the program `minepump.js` from 71% to 79%, the coverage of `alarm.js` is improved by 48.8% from 33.6% to 50%. The coverage metric can be improved much more on the other programs,

[5] `https://sv-comp.sosy-lab.org`
[6] `https://github.com/rdelhougne/Amygdala`

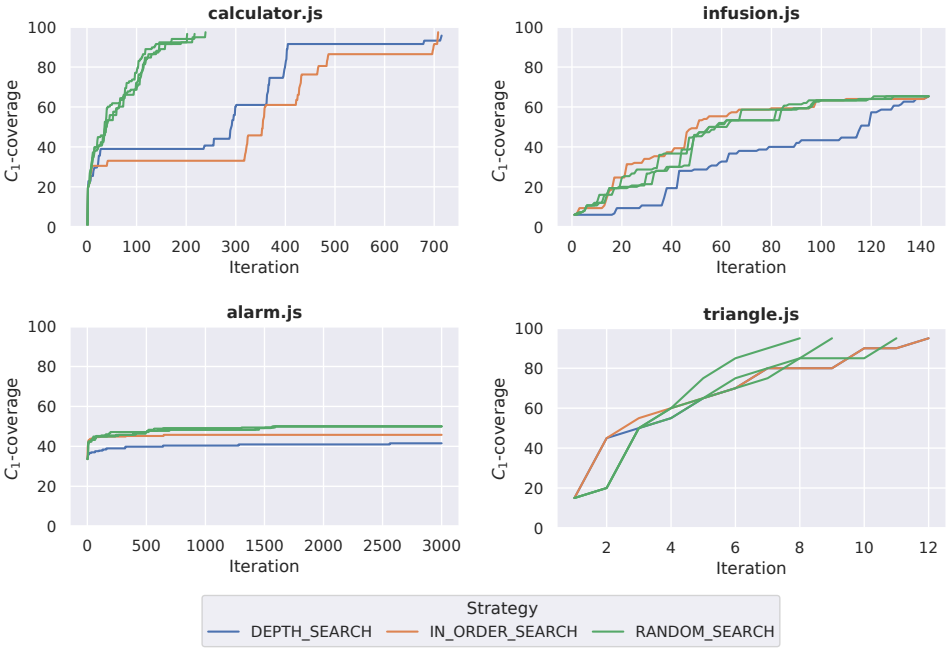


Fig. 4: Branch coverage of various test programs.

with `calculator.js` and `triangle.js` almost reaching 100% branch coverage. Notably, the tactic `RANDOM_SEARCH` always performs equal, if not better, than the other tactics. Especially with the program `calculator.js`, where it can consistently reach a high code coverage in about one-third of the iterations, or with program `alarm.js`, where the random tactic always reaches a higher code coverage, although by a small margin.

4.2 Runtime Evaluation

The total run time measurements further underline the performance of this strategy. Figure 5 shows the total runtime of the fuzzing processes. The random strategy needs needs much less time to test `alarm.js` and `minepump.js`, despite running the same number of iterations. The runtime differences in these cases were caused by path constraints that take much more time to be solved by the SMT-solver and are simply less likely to be hit by the `RANDOM_SEARCH` strategy because this strategy does not exhaustively explore the execution tree with respect to boundaries like maximum depth. This effect is particularly noticeable in the program `calculator.js`. In addition, the smaller number of iterations (cf. Figure 4) ensures that the random strategy with a runtime of 7.5 seconds only needs a fraction of the time compared to the other strategies.

Fig. 5: Total runtime of the fuzzing process for various test programs.

5 Discussion

Section 4 shows a preliminary evaluation of the fuzzer. The applicability of the fuzzer in real-world applications is, however, limited at this time. This has two leading causes: First, the fuzzer only does support a subset of the ECMAScript standard to this date. This subset includes all arithmetic, logical, and comparison operators (except for bitwise operators) and several frequently used string and math operations. Despite the extensive support of operators, the presented fuzzer is relying purely on concolic-execution. Therefore, it needs to model the whole execution flow of the tested program exhaustively. If the fuzzer or the SMT-Solver does not support just one statement and a branch depends on this statement, the path conditions for this branch cannot be constructed or solved. As a result, every part of the program lying behind this branch cannot be reliably reached and tested by the fuzzer. A solution to this problem is the extension of the support for the ECMAScript standard, which is work-intensive as explained below, or to build a *hybrid* fuzzer. Such a fuzzer uses a more straightforward method like grammar-fuzzing for most of the testing procedure and only relies on concolic/symbolic execution at difficult to reach sections of the program [MS07].

The results from chapter 3 showed that the Truffle API is an elegant way to monitor programs at runtime. The tested programs had a high execution speed despite the instrumentation due to the simultaneous optimization of AST-interpreter and tool by the GraalVM. However, the usage of wrapper-nodes imposes a drawback to this architecture: The detailed information needed to successfully implement concolic-execution forces the programmer to directly work with the low-level language implementation, instead of one of the other high-level interfaces Truffle provides. As a result, the programmer has to define a symbolic behavior for many different nodes, each of which can be arbitrarily complex. A search for class definitions for nodes in the JavaScript language implementation resulted in 634 definitions, most of which potentially can appear in an executed AST. This quantity is a lot more than the maximum of 256 opcodes of the Java VM [Li20], or the strictly defined set

of operations in LLVM-IR [Ll21]. The problem is worsened by nodes with extremely complex behavior like `JSArrayJoinNodeGen`[7]. The behavior of this node has to be manually symbolically modeled because the execution of the node is only observable as a whole. The dependency on the low-level JavaScript implementation also prevents the fuzzer from being easily ported to other languages. To make this possible, Truffles's high-level polyglot API would have to provide more specific information, which contradicts its deliberately abstract implementation.

6 Conclusion and Outlook

Due to the dynamic nature of the JavaScript language, the support for sophisticated code analysis tools lacks compared to other, more strict languages like C/C++ or Java. This work shows the implementation of a concolic-fuzzer for JavaScript using the capabilities of the Truffle API as part of the GraalVM. The implemented fuzzer extended the standard behavior of the AST interpreter to extract the required information about the tested program at runtime. The study of this approach on several test programs contained an evaluation of code-coverage metrics and a consideration of runtime measurements. The fuzzer showed promising results on the testing scenarios, reaching more than 95% branch coverage on some programs. This result is accomplished without providing any information about the input structure of the programs. This testing approach is therefore completely automated. In addition, the fuzzer is tuned to the peculiarities of JavaScript through the special analysis targets and can thus detect common errors at an early stage. While the use of GraalVM and Truffle provides a highly efficient and fast execution speed of the tested program and the fuzzer, the implementation of the fuzzer was also work-intensive due to the high number of different data structures present in the AST. Therefore, a direction of future work is the modification of the language implementation to bring the requirements of execution speed and easy-to-use instrumentation closer together.

Acknowledgement

I thank my supervisors Falk Howar and Malte Mues for the fruitful discussions and feedback during the master thesis leading to this paper.

References

[Ba18] Baldoni, R. et al.: A Survey of Symbolic Execution Techniques. ACM Computing Surveys 51/3, 50:1–50:39, 2018.

[7] `JSArrayJoinNodeGen` corresponds to the JavaScript function `Array.prototype.join()`.

[GKS05] Godefroid, P.; Klarlund, N.; Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05, ACM Press, Chicago, IL, USA, pp. 213–223, 2005.

[GLM08] Godefroid, P.; Levin, M.; Molnar, D.: Automated Whitebox Fuzz Testing. In: Proceedings of the Network and Distributed System Security Symposium. NDSS '08, The Internet Society, San Diego, CA, USA, pp. 151–166, 2008.

[Ka15] Kannavara, R. et al.: Challenges and opportunities with concolic testing. In: 2015 National Aerospace and Electronics Conference. NAECON '15, IEEE, Dayton, OH, USA, pp. 374–378, 2015.

[Li20] Lindholm, T. et al.: The Java® Virtual Machine Specification, Java SE 15 Edition, 2020.

[Ll21] LLVM Language Reference Manual, URL: `https://llvm.org/docs/LangRef.html`, visited on: 05/12/2021.

[Lu16] Luckow, K. et al.: JDart: A Dynamic Symbolic Analysis Framework. In: Tools and Algorithms for the Construction and Analysis of Systems. TACS '16, Springer, pp. 442–459, 2016.

[MS07] Majumdar, R.; Sen, K.: Hybrid Concolic Testing. In: 29th International Conference on Software Engineering. ICSE '07, IEEE, Minneapolis, MN, USA, pp. 416–426, 2007.

[OPZ11] Ocariza, F.; Pattabiraman, K.; Zorn, B.: JavaScript Errors in the Wild: An Empirical Study. In: Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering. ISSRE '11, IEEE, Hiroshima, Japan, pp. 100–109, 2011.

[PS15] Pradel, M.; Sen, K.: The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In: 29th European Conference on Object-Oriented Programming. ECOOP '15, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 519–541, 2015.

[Sa10] Saxena, P. et al.: A Symbolic Execution Framework for JavaScript. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. SP '10, IEEE, Washington D.C., USA, pp. 513–528, 2010.

[Su18] Sun, H. et al.: Efficient Dynamic Analysis for Node.js. In: Proceedings of the 27th International Conference on Compiler Construction. CC '18, ACM Press, Vienna, Austria, pp. 196–206, 2018.

[Vi03] Visser, W. et al.: Model Checking Programs. Automated Software Engineering 10/2, pp. 203–232, 2003.

[Wü13] Würthinger, T. et al.: One VM to Rule Them All. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward! '13, ACM Press, New York, NY, USA, pp. 187–204, 2013.