Software Quality Journal manuscript No.
(will be inserted by the editor)

Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing

Igor Lima · Jefferson Silva · Breno Miranda* · Gustavo Pinto · Marcelo d'Amorim

Received: date / Accepted: date

Abstract Context. JavaScript is a popular programming language today with several implementations competing for market dominance. Although a specification document and a conformance test suite exist to guide engine development, bugs occur and have important practical consequences. Implementing correct engines is challenging because the spec is intentionally incomplete and evolves frequently. **Objective.** This paper investigates the use of test transplantation and differential testing for revealing functional bugs in JavaScript engines. The former technique runs the regression test suite of a given engine on another engine. The latter technique fuzzes existing inputs and then compares the output produced by different engines with a differential oracle. **Method.** We conducted experiments with engines from five major players–Apple, Facebook, Google, Microsoft, and Mozilla–to assess the effectiveness of test transplantation and differential testing. **Results.** Our results indicate that both techniques revealed several bugs, many of which confirmed by developers. We reported 35 bugs with test transplantation (23 of these bugs confirmed and 19 fixed) and reported 24 bugs with differential testing (17 of these confirmed and 10 fixed). Results indicate that most of these bugs affected two engines–Apple's JSC and Microsoft's ChakraCore (24 and 26 bugs, respectively). To summarize, our results show that test transplantation and differential testing are easy to apply and very effective in finding bugs in complex software, such as JavaScript engines.

Keywords Test Transplantation · Differential Testing · JavaScript

* Corresponding author: Breno Miranda · E-mail: bafm@cin.ufpe.br

Igor Lima · E-mail: isol2@cin.ufpe.br
Jefferson Silva · E-mail: jefferson.alves.silva@icen.ufpa.br
Breno Miranda · E-mail: bafm@cin.ufpe.br
Gustavo Pinto · E-mail: gpinto@ufpa.br
Marcelo d'Amorim · E-mail: damorim@cin.ufpe.br

Universidade Federal de Pernambuco, Recife, PE, Brazil

1 Introduction

JavaScript (JS) is one of the most popular programming languages today [72, 67], with penetration in various software development segments including, web, mobile, and, more recently, the Internet of Things (IoT) [69]. The interest of the community for the language encourages constant improvements in its specification [78]. It is natural to expect that such improvements lead to sensible changes in engine implementations [42]. Even small changes can have high practical impact. For example, in October 2014 a new attribute added to Array objects resulted in the MS Outlook Calendar web app to fail under Chrome [19, 28].

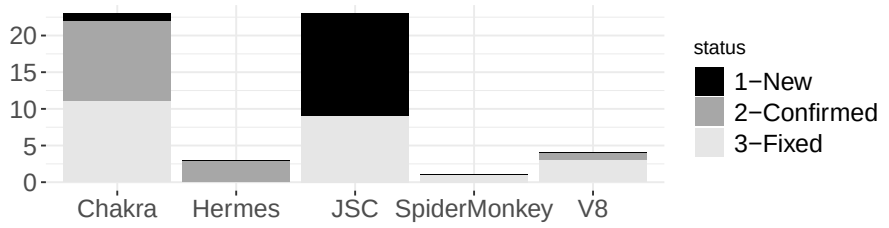
Finding bugs in JS engines is an important problem given the range of applications that could be affected with those bugs. It is also challenging. Specifications are intentionally incomplete as to enable development flexibility. In addition, they evolve frequently to accommodate the pressing demands from developers [77]. An official conformance test suite exists for JS [75], but, naturally, many test scenarios are not covered in the suite. In addition, we noticed that a significant fraction (5 to 15%) of the tests in that suite fail regularly in the most popular engines, reflecting the struggle of developers in keeping up with the pace of spec evolution (see Table 3).

This work, which is empirical in nature, reports on a study to evaluate the ability of two testing techniques to expose bugs in JavaScript engines.

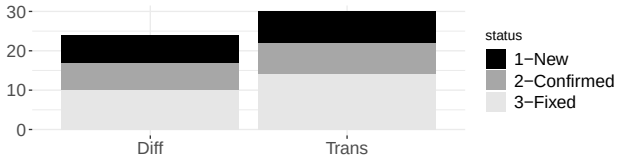
- *Test transplantation.* This technique evaluates the effect of running test files written for a given engine in other engines. The intuition is that developers design test cases with different objectives in mind. As such, replaying these tests in different engines could reveal unanticipated problems.
- *Cross-engine differential testing.* This technique fuzzes existing test inputs [55] and then compares the output produced by different engines using a differential oracle. The intuition is that interesting inputs can be created from existing inputs and multiple engines can be used to address the lack of oracles for the newly created inputs.

We selected these two techniques because they can take advantage of existing test suites and reuse them — in a semi-automated way as we propose here — to enhance the coverage of the engine under testing. Cross-engine differential testing is semi-automated as developers need to decide if alarms are manifestations of real bugs. Test transplantation is also semi-automated and developer intervention may be required to ensure that a test case from one engine can be performed on another engine without adaptations; or to verify that alarms are manifestations of real bugs and not the result of an unsupported feature or an obsolete specification. This study measures the ability of these techniques in finding bugs and the human cost associated with each technique.

Related Ideas. Differential Testing [13] (DT) has been applied in a variety of contexts to find bugs [85, 18, 9, 17, 65, 71, 87]. It has shown to be specially practical in scenarios where the observation of difference gives a strong signal



(a) Bug reports per engine.



(b) Bug reports per technique.

Fig. 1 Summary of bug reports.

of a real problem. For example, Mozilla runs JS files against different configurations of a given build of their SpiderMonkey engine (e.g., trying to enable or not eager JIT compilation[1]). A positive aspect of the approach is that it can be fully automated—as only one engine is used, the outcomes of the test in both configurations are expected to be identical. The Mozilla team uses this approach since 2002; they have been able to find over 270 bugs since then [58], including security bugs. Cross-engine differential testing, in contrast, has not been widely popularized because the reported differences are more unlikely to be false alarms. In this context, a number of legitimate reasons exist, other than a bug, for a test execution to manifest discrepancy (see Tables 5 and 8). As consequence, humans need to inspect the reports.

Results. We considered the following engines–ChakraCore (Microsoft), JavaScriptCore (Apple), V8 (Google), SpiderMonkey (Mozilla), and Hermes (Facebook). Figure 1 shows the breakdown of bug reports per engine (1a) and per technique (1b). Each stacked bar breaks down the bugs per status (e.g., "1-New"). The prefix number indicates the ordering that status labels are assigned. Several of these reports have the label "3-Fixed", indicating that bug fixes have been incorporated into the code already. Note that most of these bugs affected two engines–ChakraCore[2] and JavaScriptCore (JSC). We also reported five bugs in V8 (four confirmed), four bugs in Hermes (two confirmed),

[1] These files are created with the grammar-based fuzzer jsfunfuzz [57]. Look for option "compare_jit" from funfuzz.

[2] Microsoft announced in December 2018 that the Edge browser will be based on Chromium and ChakraCore development would be discontinued [40].

and one bugs in SpiderMonkey (one confirmed). Our results show that both techniques revealed several bugs, most of which confirmed by developers. Test transplantation revealed 35 bugs (of which, 23 were confirmed and 19 were fixed) whereas differential testing revealed 24 bugs (of which, 17 were confirmed and 10 were fixed). Overall, results indicate that both techniques were successful at finding bugs. The number of confirmed or fixed bugs are similar. Most bugs we found are of moderate severity.

Key Findings. To sum up, we found that 1) Differential testing and Test Transplantation are practical and effective techniques to find bugs on real, complex, and widely used software systems and 2) Even for problems with fairly clear specifications, as in JavaScript, there is likely (a lot of) variation between different implementations, which brings intrinsic challenges to developers that work on them. Section 8.4 expands and elaborates our key findings and lessons learned.

Contributions. The most important contribution of this work is empirical: we provide a comprehensive study analyzing the effectiveness of test transplantation and differential testing in revealing functional bugs in popular JavaScript engines. Additional contributions include: 1) A number of bugs found and fixed. We reported a total of 59 bugs. Of these, 39 bugs were confirmed and 29 bugs were fixed. 2) An infrastructure for performing test transplantation and differential testing. The source code produced and the generated data sets of tests and bugs are publicly available online at the following link:

`https://github.com/damorimRG/entente/`.

To summarize, this paper provides initial, yet strong evidence that test transplantation and differential testing are simple and effective techniques to find functional bugs in JavaScript engines and should be encouraged.

2 JavaScript

JavaScript engines are virtual machines that parse source code, compile it in bytecodes, and run these bytecodes. These engines implement some version of the ECMAScript (EcmaScript), which emerged with the goal to standardize variants of the language, such as Netscape's JavaScript and Microsoft's JScript[3]. The EcmaScript specification is regulated by Ecma International [25] under the TC39 [75] technical committee. Every year, a new version of the EcmaScript specification is released with new features and minor fixes [77, 78].

The specification of JavaScript is incomplete for different reasons. Certain parts of the specification are undefined; it is the responsibility of the community to regulate the evolution of the language. The JavaScript spec uses the label "implementation-dependent" to indicate these cases, where behavior may differ from engine to engine. One reason for such flexibility in the spec is to enable compiler optimizations. For example, the JS `for-in` loop construct

[3] The name JavaScript still prevails today, certainly for historical reasons.

Table 1 Engines selected.

Team	Name	URL	# Stars	DOB
Apple	JSC (WebKit)	[84]	3300+	Jun 2001
Google	V8	[20]	9800+	Jun 2008
Microsoft	ChakraCore	[52]	7200+	Nov 2009
Mozilla	SpiderMonkey	[59]	1100+	Mar 1996
Facebook	Hermes	[35]	5400+	Jul 2019

does not clearly specify the iteration order of elements [73, 41] and different engines capitalize on that for loop optimizations [14]. As another example, the specification states that if the `Number.toPrecision()` function is called with multiple arguments then the floating-point approximation is implementation-dependent [26]. Various other cases like these exist in the specification. Added to that, given the speed the specification changes and the complexity of the language some features are not fully implemented as can be observed by the Kangax compatibility table [42]. It is also worth noting that, as in other languages, some elements in JS have non-deterministic behavior (e.g., `Math.random` and `Date`). A test that makes decisions based on these elements could, in principle, produce different outcomes on different runs. Carefully-written test cases should not manifest this kind of flaky behavior. As previously mentioned, all those aspects make testing JS engines challenging, albeit very important given the its tremendous popularity.

3 Engines Studied

We selected JS engines according to the following criteria: 1) Released latest version after Jan 1, 2018, 2) Contains more than 1K stars on GitHub, and 3) Uses a public issue tracker. We looked for highly-maintained (as per the first criterion) and popular (as per the second criterion) engines. As we wanted to report bugs, we also looked for project with public issue trackers. We initially selected four JS engines: JSC, V8, ChakraCore, and SpiderMonkey. Later, we included Hermes in the list of studied engines. The main reason was to investigate how our approach would work on a recently introduced JS engine. More about Hermes on Section 7. Table 1 lists the engines we analyzed. It is worth noting that we used Google Chrome Lab's JSVU tool [32] to automatically install and configure versions of different JS engines in our host environment. This is important as we aim to use the most recent stable versions of each engine as to avoid reporting old and already-fixed bugs to developers.

4 Mined JS Files

Obtaining good sets of JS test cases is imperative to evaluate the techniques in this paper. For that, we looked for JS files from various sources: 1) test files

Table 2 Number of test files. Dashed rectangle under column "type-in-all" shows the tests used for test transplantation whereas the rectangle under column "no-fail-in-all"—a subset of the "type-in-all" tests—shows the tests used in cross-engine differential testing.

Name	Source	# JS files			
		total	pass-in-par.	type-in-all	no-fail-in-all
Test262	[74]	31,276	-	29,846	17,639
JSC	[84]	1,265	1,130	1,122	1,054
SpiderMonkey	[83]	3,122	2,155	2,103	1,837
V8	[20]	1,084	482	478	426
Hermes	[35]	1,728	680	661	632
Duktape	[23]	1,195	1,195	921	915
JerryScript	[39]	1,951	1,951	1,878	1,837
JSI	[38]	99	99	63	63
Tiny-js	[82]	49	49	37	37
Babel	[10]	9,953	-	2,198	1,745
BlogEngine.NET	[11]	954	-	24	18
		52,676	7,741	38,209	26,203

from the Test262 [75] conformance suite of the ECMA262 specification [78], 2) test files from the test suite of our selected engines; these files are accessible from the engine's official repositories, 3) test files from the suites of other public engines (i.e., Duktape, JerryScript, JSI, Tiny-js, Babel, and BlogEngine.NET), and 4) test files mined from issue trackers of these engines.

Table 2 shows the breakdown of tests. Column "Name" and "Source" show the origin of the test suite. Column "total" shows the number of test cases associated with a given source of JS files. Column "pass-in-par." shows the number of test cases that pass in the corresponding engine. We discarded tests that fail in their engine as we could not reliably indicate the reason for the failure, so we assumed the test could be broken. We removed 63 test cases that fail for that reason–6 tests from JSC and 57 tests from SpiderMonkey.

Column "type-in-all" shows the number of test cases whose executions do not throw dynamic type errors in any of the engines because of an undefined variable or property. These cases were captured by looking for the presence of `ReferenceError` and `TypeError` on the output. A `ReferenceError` (respectively, `TypeError`) is raised when test execution attempts to access an undefined variable (respectively, property of an object). We discarded those tests to avoid noise in the experiments as they indicate some missing feature in the implementation of the engine as opposed to bugs. For example, some tests use non-portable names (e.g., JSC's `drainMicrotasks()` and SpiderMonkey's `Error.lineNumber`) or use functions that, albeit part of the spec, not all engines currently support. Similarly, the ChakraCore project does not use common assertions available used in JS program, which other JS engines support. Instead, it uses its own testing framework, WScript. In such cases, we are unable to reproduce the ChakraCore unit tests in other engines—it raises a `ReferenceError`. Also, updating the other engines to use WScript would require a non-trivial manual effort. This is why ChakraCore does not appear in

Table 2. For the evaluation of test transplantation, we used the 9,485 tests included in the dashed rectangle under column "type-in-all", i.e., all tests under that column but the Test262 tests. We did not consider tests from the conformance suite as they are more likely to indicate missing features as opposed to bugs. In addition, engine developers have access to these tests and are encouraged to run them. Column "no-fail-in-all" shows the tests for which all engines pass. The tests in this set are used as fuzzing seeds in the evaluation of differential testing. The guarantee that tests pass in all engines assures that discrepancies are related to the changes in the input produced by fuzzers.

Cleansing We noticed that some of the tests we found depend on external libraries, which not all selected engines support. We decided to discard those. For example, we found many tests based on Node.js [81] that require libraries to be installed before running the test and different tests require different sets of libraries. Supporting those tests would require an extra setup step and would slow down the execution of our experiments. Also, as already mentioned, we did not consider tests from the ChakraCore repository because they depend on non-portable objects.

Test Harness We noticed that some engines use a custom shell to run tests, including a harness with specific assertions. For example, tests provided by Mozilla contain a lot of custom functions (e.g., `assertThrowsInstanceOf`, `assertEqArray`, and `getPromiseResult`) and those are included in the shell to make the test run correctly. For that, we needed to make minor changes in the testing infrastructure to be able to run the tests uniformly across all engines. More precisely, we needed to mock non-portable functions, which are only available in certain engines. Since we had to transplant more than 40k test files, we tried to mock the minimum amount of code possible to make the test file work in the other engine. For instance, if a test had a statement such as `print(1==1);` we manually refactored that statement to `assert(1==1)` which should throw an assertion violation if the condition returns false. If we found the change would require substantial manual work, we opted to discard the test instead.

Dedup The number of tests in V8 is low because we discarded duplicate tests with Mozilla and JSC. The rationale is to avoid inflating results and giving credit where it is due. We also wrote a script that compares each pair of tests from different suites for similarity. We did not find identical tests, although it is possible there are equivalent tests modulo renaming.

4.1 Mining Tests From Issue Trackers

Test cases embedded in issue trackers are an important source of data as they may have already shown useful to reveal problems in some engine. For example, a developer may have found a corner case that a given engine did not handle properly. Therefore, it is possible that some other engine does not handle that case as well. For that reasons, we thought that we should not ignore

issue trackers. By manually analyzing a sample of issues, we observed that developers either 1) add test cases as attachments of an issue or 2) embed the test cases within the textual description of an issue. The test cases in attachments are longer compared to the test cases embedded in issue descriptions whereas the latter are more common. Consequently, we thought we should handle both cases.

To obtain test files included as attachments, we wrote a crawler to visit the issue trackers of all engines listed in Table 1 and we were able to retrieve a total of 490 files. To mine tests from the textual descriptions we proceeded as follows. First, we broke the text describing the issue in paragraphs and used a binary classifier to label each paragraph as "code" or "not code" (i.e., text written in natural language). Then, based on that information, we merged consecutive paragraphs labeled as "code" and used a JS parser to check well-formedness of the retrieved code fragment. Using that method we were able to retrieve a total of 1,240 additional files. All those files were included in Table 2.

For the classification of "code" vs. "nocode", we used popular techniques for solving NLP classification problems [45]. First, we used word2vec [54], a popular NLP technique to produce word embeddings. A word embedding is a mapping of words to vectors of real numbers. Then, we used a multi-layer perceptron [68] to infer the probability of the input belonging or not to the class based on the distance between sentences (i.e., the distance from an input sentence to a code example or to a nocode example) as induced by the distance of comprising words computed with word2vec. The classifier labels the input as code if the predicted probability of the input being code is 0.7 or higher. We used a corpus with 25K samples of English paragraphs and 25K snippets of JS code to train and test the classifier and obtained an accuracy of 98%. This classifier is publicly available from our website as a separate component.

5 Cross-Engine Differential Testing

This section describes the infrastructure we used for cross-engine differential testing. Figure 2 illustrates the workflow of the approach. It takes on input a list of JS files and generates warnings on output. Numbered boxes in the figure denote the data processors and arrowed lines denote data flows. The cycle icons indicate repetition–the cycle icon close to the cylinder icon indicates that each file in the input list will be analyzed in separate whereas the other cycle icon shows that a single file will be fuzzed multiple times.

The bug-finding process works as follows. First, for a given test input, the toolchain produces new inputs using some off-the-shelf input fuzzer (step 1). Section 5.3 describes the fuzzers we selected. Then, the oracle checks whether or not the output produced for the fuzzed file is consistent across all engines (step 2). In case the test passes in all engines or fails in all engines (i.e., the output is consistent), the infrastructure ignores the input. Otherwise, it considers the input as potentially fault-revealing; hence, interesting for human inspection. Finally, to facilitate the human inspection process, the infrastruc-

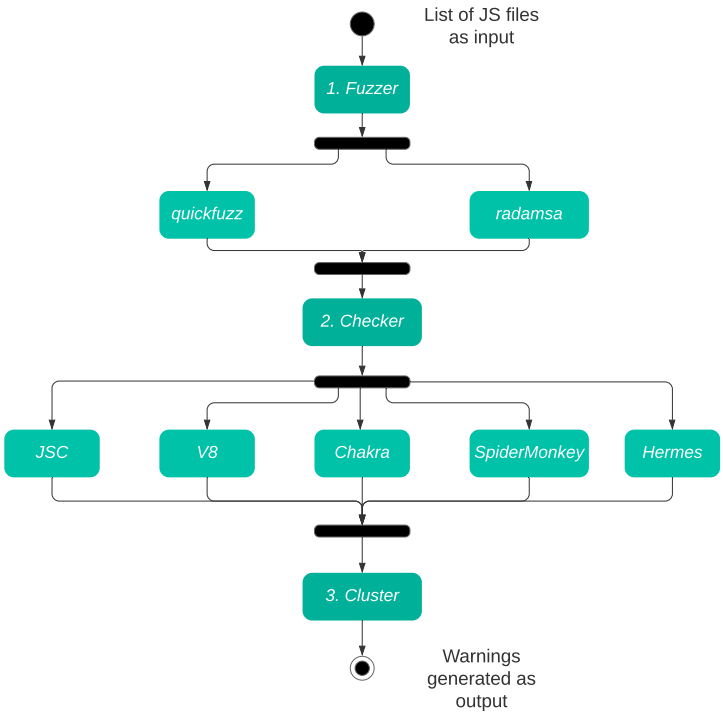


Fig. 2 The workflow of the referred differential testing approach.

ture prioritizes warnings and clusters them in groups (step 3). We describe these features in Sections 5.1 and 5.2. Note that a number of reasons exist, other than a bug, for discrepancy to arise (see Tables 5 and 8) and there is no clear automatic approach to precisely distinguish false and true positives. As such, a human needs to inspect a warning to classify the issue. As mentioned earlier, this justifies why differential testing is challenging to automate at the functional level. Existing techniques that use differential testing deal with false alarms differently. For example, Mozilla varies the configurations of their SpiderMonkey engine, but the implementation is the same [60]. Non-determinism is therefore more likely to be associated with the variations across version as these version use the same core implementation. CSmith [86] addresses the problem by trying to avoid generating C files that produce undefined behavior. Mapping all sources of undefinedness in JS is impractical.

For step 2, we considered using the open-source tool eshost-cli [12], also used at Microsoft, for checking output discrepancy. However, we noticed that eshost-cli does not handle discrepancies involving crashes. Our tool supports the execution of the binaries themselves to obtain these fatal errors. It is important to note that our checker does not support the case where the test fails in all engines with a different kind of failure as it is unclear how to

properly relate those failures. Currently, our infrastructure does not report discrepancy on that case. We left that as future work as we already found several discrepancies even without that.

5.1 Prioritization

We prioritized warnings based on their likelihood of manifesting a real bug. We defined two types of warnings based on empirical evidence we obtained while analyzing bugs. The types "hi" and "lo". Warnings of the kind "hi" are associated with the cases where the test code executes without violating any internal checks, but it violates an assertion declared in the test itself or its harness. The rationale is that the test data is more likely to be valid in this case as execution does not raise exceptions in application code. Warnings of kind "lo" cover the remaining cases. These warnings are more likely to be associated with invalid inputs. They reflect the cases where the anomaly is observed during the execution of application functions as opposed to assertions. We observed that different engines often check pre-conditions of functions differently. It can happen, for example, that one engine enforces a weaker pre-condition, compared to another engine, on the inputs of a function and that is acceptable. In those cases, the infrastructure would report a warning that is more likely to be associated with an invalid input produced by the fuzzer, i.e., it is likely to be a "bug" in the test code as opposed to a bug in the engine. Recall that, for differential testing, we only use seed tests that pass in all engines.

```
var buffer = new ArrayBuffer(64);
var view = new DataView(buffer);
view.setInt8(0,0x80);
assert(view.getInt8(-1770523502845470856) === -0x80);

Message from Engines (1:V8, 2:JavaScriptCore, 3:SpiderMonkey):
1. RangeError: Offset is outside the bounds of the DataView
2. RangeError: byteOffset cannot be negative
3. RangeError: invalid or out-of-range index
```

Fig. 3 Example of a "lo" warning that led to a confirmed bug report in ChakraCore. The bug is caused by a required precondition check in the implementation of function `ToIndex` [79], which is indirectly called by the test.

Despite the problem mentioned above, "lo" warnings can reveal bugs. Figure 3 shows one of these cases. In this example, the test instantiates an `ArrayBuffer` object and stores an 8-bit integer at the 0 position. According to the specification [79], a `RangeError` exception should be thrown if a negative value is passed to the function `ToIndex`, indirectly called by the test case from the function call `getInt8()`. In this case, however, the ChakraCore engine did not throw any exception, as can be confirmed from the report that our infrastructure produces starting with text "Engine Messages" at the bottom of Figure 3. This is a case of undocumented precondition. It was fixed by developers and is no longer present in the most recent release of the engine.

5.2 Clusterization

Clusterization is complementary to prioritization. It helps to group similar warnings reported by our infrastructure. We only clustered "lo" warnings as "hi" warnings produce messages that arise from the test case, which are typically distinct. Figure 3 shows a test that was originally available in the JSC suite. The radamsa fuzzer mutated the test. It introduced the negative long `-177052350284547085 6` as a parameter of the `view.getInt8()` method. Before the mutation, the `view.getInt8()` method received zero as parameter. At the bottom of this figure, there is a sequence of three elements that we use to characterize a warning: 1) the identifier of an engine, 2) the exception it raises, and 3) the message it produces on a "lo" warning. This sequence of triples defines a warning signature that we use for clustering. It is worth mentioning that we filter references to code in messages as to increase ability to aggregate warnings. Any warnings, including this one, that has this same signature will be included in the same "bucket". Considering this particular example, the signature for that cluster will be [(JavaScriptCore, "RangeError", "byteOffset cannot be negative"), (SpiderMonkey, "RangeError", "invalid or out-of-range index"), (V8, "RangeError", "Offset is outside the bounds of the DataView")].

5.3 Fuzzers

Fuzzers are tools for generating inputs for a given input format [51]. Different fuzzing strategies exist. We analyzed *generational* and *mutational* fuzzers.

Generational fuzzers create inputs anew, typically following a language description, typically context-free grammars. Intuitively, those fuzzers implement a traversal of the production rules of a grammar to create syntax trees, which are then pretty-printed and used as a fresh input. Such strategy to create inputs produces inputs that are syntactically valid by construction. We analyzed four grammar-based fuzzers–Grammarinator [36], jsfunfuzz [57], LangFuzz [37], and Megadeth [33]. Unfortunately, none of those were effective out-of-the-box. For example, we produced 100K inputs with Grammarinator and only few inputs were semantically valid. With Megadeth, we were able to produce more valid inputs as it contains some heuristics to circumvent violations of certain typing rules. Nonetheless, running those inputs in our infrastructure we were unable to find discrepancies. Inspecting those inputs, we realized that they reflected very simple scenarios. To sum up, a high percentage of inputs that Grammarinator and Megadeth generated were semantically-invalid that we needed to discard whereas the valid inputs manifested no discrepancies. Considering jsfunfuzz [57], which has been developed by the Mozilla team, we noticed that, in addition to the issues mentioned above, it produces inputs that use functions that are only available in the SpiderMonkey engine. We would need either to mock those functions in other engines or to discard those tests. Considering LangFuzz [37], the tool is not publicly available. Another fundamental issue associated with generational fuzzers in our context is that

the tests they produce do not contain assertions; to enable the integration of this kind of fuzzers in our infrastructure—we would need to look for discrepancies across compiler error messages as opposed to assertion violations. All in all, although grammar-based fuzzers have been shown effective to find real bugs [37], we did not consider those fuzzers in this study for the reasons above.

Mutational fuzzers modify inputs files provided as seeds. Gray-box mutational fuzzers use coverage information to guide the mutation process. American Fuzz Loop (AFL) [1] and libFuzzer [3] are the most popular coverage-guided fuzzers of today. These fuzzers run tests inputs against instrumented versions of the programwith the typical goal of finding universal errors, like crashes and buffer overflows. The instrumentation adds code to collect branch coverage and to monitor specific properties[4]. AFL and libFuzzer work very similarly. They use coverage to find inputs that uncover a new branch and hence should be fuzzed more. These tools take as input a program binary (say, a JS engine), which is instrumented to collect coverage information and to capture runtime violations (e.g., illegal memory accesses), and one seed input to that program (say, a JS program) and produces on output fault-revealing inputs. We chose to use AFL and instrument V8. Unfortunately, we found that most of the inputs produced by AFL violate the JS grammar. We would need to translate production rules of Python to the AFL format to circumvent that issue. We found that the fuzzing task can take days for a single seed input and there is no clear way to guide the exploration. That happens because the fuzzer aims to explore the entire decision tree induced from the engine's main function, including the branches associated with the higher layers of the compiler (e.g., lexer and parser). One way to mitigate that problem is by writing fuzzing targets (a.k.a. targets) for specific program functions. Although that approach has shown to be effective at Google [4, 2], it requires domain knowledge to create the calling context to invoke the fuzz target. For that, we decide not to consider coverage-based in this study.

We used two black-box mutational fuzzers in this study: radamsa [34] and quickfuzz [70]. These fuzzers require no instrumentation and domain knowledge. They mutate existing inputs randomly. The strength of the approach is limited by the quality of the test suite and the supported mutation operators, which are typically simple. We chose these specific fuzzers because, conceptually, one complements the other. quickfuzz creates mutations like radamsa. However, in contrast to radamsa, quickfuzz is aware of the JS syntax; it is able to replace sub-trees of the syntax tree [33] with trees created anew. Notwithstanding, since radamsa performs simpler syntactic source code modifications, it also produces a higher number of valid inputs.

[4] There are options in the clang toolchain to build programs with fuzzing instrumentation [3]. clang provides several sanitizers for property checking [50].

6 Results

The goal of this paper is to assess ability of test transplantation and differential testing to find functional bugs in JavaScript engines. Based on that, we pose the following three questions:

RQ1. How conformant are the engines to the Test262 suite?
RQ2. How effective is test transplantation to find bugs?
RQ3. How effective is cross-engine differential testing to find bugs?

The first question focuses on the conformance of our selected engines to the official Test262 suite [74] (Section 6.1). In the limit, bugs would have low relevance if the engines are too unreliable. The second question focuses on the effectiveness of test transplantation (Section 6.2). The rationale for using inputs from different engines is that developers consider different goals when writing tests—suites written for a given engine may cover scenarios not covered by a different engine. The third question evaluates the effectiveness of cross-engine differential testing to find bugs (Section 6.3). The rationale for this question is that fuzzing inputs may explore scenarios not well-tested by at least one of the engines.

6.1 Answering RQ1 (Test262 Conformance)

The ECMA Test262 [74] test suite serves to check conformance of engines to the JS standard. It is acceptable to release engines fulfilling the specification only partially [42]. We expect that the pass rate on this suite provide some indication of the engine's maturity. In the limit, it is not desirable to flood bug reports on engines at early stages of development. For this experiment, we ran the suite once a day for seven consecutive days and averaged the passing ratios. We performed seven consecutive executions because, since the studied JS engines release new versions on a daily basis, we wanted to make sure if the failing tests raised by the Test262 suite were rapidly fixed by the maintenance team. If errors were quickly fixed, this would suggest that a given engine would be closely aligned with the ECMAScript specification. Table 3 shows the average number of passing tests over this period. The variance of results was negligible; for that reason, we omitted standard deviations. We noticed that all engines but ChakraCore used some variant of the Test262 suite as part of their regression process. We used the same version in this experiment [74].

Results show that there are still many unsupported scenarios as can be observed from the percentages in the table. The number of passing tests is high and similar for JSC, V8, and SpiderMonkey. Moreover, one can also note that Hermes and ChakraCore have a low passing ratio in this test suite. Interestingly, ChakraCore is also the one we were able to find more bugs (as per Figure 1). Although it is plausible to find correlation between the passing ratios and reliability as measured by the number of bugs found, we do not imply causality. As discussed above, it is important to note that failures in

Table 3 Percentage of passing tests on the Test262 conformance suite.

engine	% passing
V8	95%
SpiderMonkey	93%
JSC	92%
ChakraCore	75%
Hermes	26%

this conformance test suite indicates missing features as opposed to bugs. Finally, since Hermes has a very low adherence to the Test262 conformance suite, we opted to conduct a case study with them. Therefore, we will only discuss Hermes data at Section 7.

Summary: Most of the engines seem to adhere well to the JS standard. Except for Hermes and ChakraCore, the passing ratio of all engines is above 90%.

6.2 Answering RQ2 (Test Transplantation)

This section reports results of test transplantation. More specifically, we analyzed the failures observed when running a test suite original from a given engine in another engine. Intuitively, we want to assess how effective is the idea of cross-fertilization of testing knowledge among JS developers.

6.2.1 Methodology

In this experiment, a developer with experience in JS analyzed each test failure, affecting a particular engine, and classified that failure as potentially fault-revealing or not. The authors supervised the classification process to validate correctness. For the potentially fault-revealing cases, one of the authors inspected the scenario and, if agreed on the classification, reported the bug to the issue tracker of the affected engine.

Table 4 Number of failures with Test Transplantation.

test suite\engine	JSC	V8	SpiderMonkey	ChakraCore
JSC	-	10	10	59
V8	41	-	3	5
SpiderMonkey	218	107	-	281
Duktape	0	4	4	1
JerryScript	23	25	22	23
JSI	0	0	0	0
Tiny-js	0	0	0	0
total	282	146	39	369

6.2.2 Results

Table 4 shows the number of failures observed for each pair of test suite and engine. The first column shows the test suites and the first row shows the engines that run those tests. We use a dash ("-") to indicate that we did not consider the combinations that run the test suite of an engine in itself. Failures in those cases would either indicate regressions or flaky tests as opposed to unknown bugs for that engine. As explained in Section 4, we used a total of 9,485 tests in this experiment. These tests are included in the dashed rectangle under column "type-in-all" on Table 2. Running those tests we observed a total of 836 failures manifested across 612 distinct files (9.2% of total). Table 4 shows that SpiderMonkey was the engine that failed the least whereas ChakraCore was the engine that failed the most. The SpiderMonkey test suite also revealed more failures than any other, perhaps as expected, given that it is the suite with more tests (see Table 2).

In particular, we were not able to reuse the SpiderMonkey tests on the Hermes engine. This happened because SpiderMonkey has its own assertion framework, but Hermes did not interpret these assertions, resulting in failures in all test executions from this engine. These failures did not happen when the SpiderMonkey tests were transplanted to the other engines, though.

The sources of false positives found in this experiment are as follows:

Undefined Behavior. False positives of this kind are manifested when tests cover implementation-dependent behavior, as defined in the ECMA262 specification [78]. For example, one of the tests from JerryScript uses the function `Number.toPrecision([precision])`, which translates a number to a string, considering a given number of significant digits. The floating-point approximation of the real value is implementation-dependent, making that test to pass only in ChakraCore.

Timeout/OME[5]. False positives of this kind typically manifest when the engine that runs the test does not optimize the code as the original engine of the test. As result, the test fails to finish at the specified time budget or it exceeds the memory budget. For example, a test case from JSC defines a function with a tail-call recursion. The test fails in all engines but JSC, which implements tail-call optimization.

Not implemented. False positives of this kind manifest when a test fails because it covers a function that is part of the official spec, but is not implemented in the target engine yet. For example, at the time of writing, ChakraCore did not implement by default various properties from the `Symbol` object. These properties are only available activating the ES6 experimental mode with the flag `-ES6Experimental`.

Non-Standard Element. These cases manifest when a function or an object property is undefined in the execution engine but we were unable to capture that by looking for error types like `ReferenceError` and `TypeError`.

Other. This category includes other sources of false positives. For example, it includes the cases where the test was valid for some previous version of the spec but is no longer valid for the current spec.

Table 5 Distribution of False (FP) and True Positives (TP).

	source	#
FP	Undefined Behavior	204
	Timeout/OME	23
	Not Implemented	54
	Non-Standard Element	122
	Other	174
TP	Duplicate	11
	Bug	24

Table 5 shows the distribution of False Positives (FPs) and True Positives (TPs). The sum of the numbers in this table corresponds to the number of files that manifested failures, i.e., 612. Considering false positives, "Undefined Behavior" was the most predominant source. Considering true positives, we found a reasonable number of duplicate reports, but not high enough to justify attempting to automate the detection of duplicates.

Table 6 lists all bugs we found with test transplantation. The first column shows the identifier we assigned to the bug, column "Engine" shows the affected engine, column "Status" shows the status of the bug report at the time of the writing. The status string appears in bold face for status "Confirmed" or higher, i.e., "Assigned" and "Fixed". Column "Severity" shows the severity of confirmed bugs, and, finally, column "Suite" shows the name of the engine that originated the test. Considering severity levels, we found that JSC [8] and SpiderMonkey [60] developers use five levels, whereas ChakraCore [53] and V8 [80] developers use only three. As usual, the smallest the number the highest the severity of the bug. We use a dash ("-") in place of the severity level for the cases where the bug report is pending confirmation. Of the 35 bugs we reported in this experiment, 23 were promoted from the status "New" to "Confirmed". Of these, 16 are severity-2 bugs. Although we did not find any critical bugs, most of the bugs are seemingly important as per the categorization given by engineers. Analyzing the issue tracker of ChakraCore, we found that severity-1 bugs are indeed rare. Considering the number of bug reports confirmed by developers, ChakraCore was the engine with the highest number: 12, with 3 bugs fixed. Considering the remaining engines, V8 developers confirmed the three bugs we reported, fixing two. Curiously, Google engineers confirmed the issued bug reports in a few hours. Likewise, we reported only one bug on Mozilla's SpiderMonkey, which was quickly fixed. Overall, we found that development teams of other engines, specially JSC, took much longer to analyze bug reports as can be observed in the JSC stacked bar from Figure 1a. However, once the team confirmed those bugs they were then quickly fixed.

Table 6 List of bugs reports from Test Transplantation.

#	Engine	Version	Status	Severity	Suite
1	JSC	606.1.9.4	New	-	JerryScript
2	ChakraCore	1.9	**Confirmed**	2	SpiderMonkey
3	ChakraCore	1.9	**Fixed**	2	SpiderMonkey
4	ChakraCore	1.10-beta	**Confirmed**	2	SpiderMonkey
5	JSC	606.1.9.4	New	-	SpiderMonkey
6	JSC	606.1.9.4	New	-	SpiderMonkey
7	JSC	606.1.9.4	**Fixed**	2	SpiderMonkey
8	ChakraCore	1.10-beta	**Fixed**	3	SpiderMonkey
9	ChakraCore	1.10-beta	**Fixed**	2	JSC
10	ChakraCore	1.10-beta	**Fixed**	2	SpiderMonkey
11	ChakraCore	1.11-beta	**Fixed**	2	JSC
12	ChakraCore	1.11-beta	**Confirmed**	2	JerryScript
13	ChakraCore	1.10.1	**Fixed**	2	SpiderMonkey
14	JSC	233840	Duplicated	2	JerryScript
15	ChakraCore	1.10.1	**Fixed**	2	JerryScript
16	ChakraCore	1.10.1	**Fixed**	3	JerryScript
17	JSC	234555	**Fixed**	2	JerryScript
18	ChakraCore	1.10.1	**Fixed**	3	JerryScript
19	JSC	234654	**Fixed**	2	JerryScript
20	V8	7.0.181	**Fixed**	3	JerryScript
21	JSC	234689	New	-	JerryScript
22	V8	7.0.237	WontFix	2	Duktape
23	ChakraCore	1.10.2	**Fixed**	2	SpiderMonkey
24	JSC	235121	**Fixed**	2	SpiderMonkey
25	JSC	235121	**Fixed**	2	SpiderMonkey
26	V8	7.0.244	**Fixed**	2	SpiderMonkey
27	ChakraCore	1.10.2	**Fixed**	2	SpiderMonkey
28	JSC	235121	New	-	SpiderMonkey
29	ChakraCore	1.11.19	**Confirmed**	-	Babel
30	JSC	262693	New	-	Babel
31	JSC	262693	New	-	Babel
32	SpiderMonkey	77.0b9	**Fixed**	3	Hermes

Summary: Test transplantation was effective at finding functional bugs. Although the cost of classifying failures was non-negligible, the approach revealed several non-trivial bugs in three of the four engines we analyzed.

6.3 Answering RQ3 (Differential Testing)

This section reports the results obtained with cross-engine differential testing.

6.3.1 Methodology

The experimental methodology we used is as follows. As explained on Section 5.3, we used Radamsa [34] and QuickFuzz [70] for fuzzing. To avoid experimental noise, we only fuzz test files that pass in all engines–a total of 26,203 tests satisfy this restriction. Those tests appear under the column "no-fail-in-all" on Table 2. We want to avoid the scenario where fuzzing produces a fault-revealing input based on a test that was already revealing failures on some engine. This decision facilitates our inspection task; it helps us establish cause-effect relationship between fuzzing and the observation of discrepancy.

We configured our infrastructure (see Figure 2) to produce 20 well-formed fuzzed files per input file, i.e., the number of fuzzing iterations can exceed the number above as we discard generated files that are syntactically invalid.

Exploratory Phase For the first three months of the study, our inspection process was exploratory. In this phase, we wanted to learn whether or not black-box fuzzers could reveal real bugs and how effective was the hi-lo warning classification. We expected the number of warnings to increase dramatically compared to the previous experiment and, if we realized that the ratio of bugs from lo warnings was rather low, we could focus our inspection efforts on hi warnings. To run this experiment, we trained eight students in analyzing the warnings that our infrastructure produced. The students were enrolled in a graduate-level testing class. We listed warnings in a spreadsheet and requested the students to update an "owner" column indicating who was working on it, but we did not enforce a strict order on the warnings the students should inspect. Recall from Section 5.2 that we clustered lo warnings in buckets. For that reason, we only listed one lo warning per representative class/bucket in the spreadsheet. First, we explained, through examples, the possible sources of false alarms they could find and then we asked the students to use the following procedure when finding a suspicious warning. Analyze the parts of the spec related to the problem and, if still suspicious, look for potential duplicates on the bug tracker of the affected engine using related keywords. If none was reported, indicate in the spreadsheet that that warning is potentially fault-revealing. We encouraged students to use lithium [5] to minimize long test cases. A bug report was filed only after one of the authors reviewed the diagnosis. Each student found at least one bug using this methodology.

Non-Exploratory Phase Results obtained in the exploratory phase confirmed our expectations that most of the bugs found during the initial period of investigation were related to hi warnings. For that reason, we changed our inspection strategy. This time, some of the co-authors inspected the bugs using a similar discipline as before. However, the set of warnings inspected and the order of inspection changed. We restricted our analysis to hi warnings and, aware that we would be unable to analyze each and every warning reported, we grouped those warnings per engine, analyzing each group in a round-robin fashion. At each iteration, we analyzed five warnings in each group. A warning belongs to the group of a given engine if only that engine manifests distinct behavior, i.e., it produces a distinct output compared to others. We separated in a distinct group the warnings for which two engines diverge. The rationale for this methodology was to give attention to each engine more uniformly, enabling more fair comparison across engines.

6.3.2 Results

Table 7 shows statistics of hi warnings. The table breaks down hi warning by the affected engine, i.e., the engine manifesting distinct output among those analyzed. Column "+1" shows the cases where more than one engine disagree

Table 7 Number of hi warning reports per engine.

fuzzer\engine	JSC	V8	ChakraCore	SpiderMonkey	+1
radamsa	151	50	331	94	528
quickfuzz	83	63	351	21	403
total	234	113	682	115	931

Table 8 Distribution of False (FP) and True Positives (TP).

		radamsa	quickfuzz
FP	Undefined Behavior	42	16
	Timeout/OME	30	15
	* Invalid Input	46	55
	* Error Message Mismatch	41	12
TP	Duplicate	36	28
	Bug	16	7

on the output. Note from the totals that the ordering of engines is consistent with the one observed on Table 4, with ChakraCore and JSC in first and second places, respectively, in number of warnings.

Table 8 shows the distribution of false positives per source. The sources of imprecision are as defined in Section 6.2 with the addition of two new sources, which we did not observe before. These new sources are marked with a "*" in the table. The source "Invalid Input" indicates that the test input violated some part of the specification. For example, the test indirectly invoked some function with unexpected arguments; this happens because fuzzing is not sensitive to function specifications. Consequently, it can replace valid with invalid inputs. The source "Error Message Mismatch" corresponds to the cases where the fuzzer modifies the assertion expression (e.g., some string expression or regular expression).

Table 9 shows the list of bugs we reported. The table shows the fuzzing tool used ("Fuzzer"), the JS engine affected ("Engine"), the status of the bug report ("Status"), the severity of the bug report ("Sev."), the priority that we assigned to the warning that revealed the bug ("Priority"), and the test suite from the original test input ("Suite"). So far, 16 of the bugs we reported were confirmed, ten of which were fixed. Note that one bug report that we submitted was rejected on the basis that the offending JS file manifested an incompatibility across engine implementations that was considered to be acceptable. As of now, we did not find any new bugs on SpiderMonkey; the bugs we found were duplicates and were not reported. For V8, we reported 2 bugs, all of them confirmed, with 1 fixed.

Summary: Cross-engine differential testing was effective at finding JS engines bugs, several of which have been fixed already.

Table 9 List of bugs reports from Differential Testing.

#	Fuzzer	Engine	Version	Status	Sev.	Priority	Suite
1	radamsa	ChakraCore	1.9	**Fixed**	2	**lo**	JSC
2	radamsa	ChakraCore	1.9	WontFix	-	hi	JSC
3	radamsa	JSC	606.1.9.4	New	-	hi	JSC
4	radamsa	ChakraCore	1.9	**Fixed**	2	hi	JerryScript
5	radamsa	JSC	606.1.9.4	**Fixed**	2	hi	JerryScript
6	radamsa	ChakraCore	1.10-beta	**Confirmed**	2	hi	TinyJS
7	radamsa	JSC	606.1.9.4	New	-	hi	TinyJS
8	radamsa	JSC	606.1.9.4	**Fixed**	2	**lo**	SpiderMonkey
9	radamsa	ChakraCore	1.10-beta	**Confirmed**	3	hi	JerryScript
10	radamsa	ChakraCore	1.10-beta	**Fixed**	2	hi	V8
11	radamsa	JSC	606.1.9.4	**Fixed**	2	hi	JSC
12	radamsa	JSC	606.1.9.4	**Fixed**	2	hi	JerryScript
13	quickfuzz	JSC	606.1.9.4	**Fixed**	2	hi	JerryScript
14	quickfuzz	ChakraCore	1.11-beta	**Confirmed**	2	hi	JerryScript
15	radamsa	ChakraCore	1.10.2	**Fixed**	3	hi	Test262
16	radamsa	V8	7.0.244	**Fixed**	2	hi	Test262
17	quickfuzz	JSC	235121	New	-	hi	Test262
18	quickfuzz	V8	7.0.244	**Confirmed**	2	hi	Test262
19	quickfuzz	ChakraCore	1.10.2	**Confirmed**	2	hi	Test262
20	quickfuzz	JSC	235318	New	-	hi	Test262
21	quickfuzz	JSC	235318	New	-	hi	Test262
22	radamsa	ChakraCore	1.11.6.0	**Confirmed**	-	hi	SpiderMonkey
23	radamsa	ChakraCore	1.11.19	**Confirmed**	-	lo	Hermes

Data Availability. The data, including the tests, warning reports, and diagnostic outcomes, is publicly available from a preserved repository `https://github.com/damorimRG/entente/`.

7 Case Study: Hermes

This section describes the experiment we conducted to evaluate the effectiveness of the techniques we studied to find new bugs in Hermes, a JavaScript engine that has started development very recently. Hermes is a JavaScript engine introduced by Facebook in 2019. According to its website, Hermes is "a JavaScript engine optimized for fast start-up of React Native apps on Android" [35]. Currently, Hermes is used as a beta component of the react-native framework, which is also maintained by Facebook. The goal of Hermes at Facebook is to increase the performance of Android applications, such as startup time, to reduce memory consumption and application size [30].

As a preparatory step to apply test transplantation and differential testing to Hermes, we ran the Test262 conformance suite on the most recent version of Hermes, 0.5.0. We observed that only 26% of the Test262 tests passed. When taking a closer look, we noticed that this high percentage of failing tests was due to the high number of features not yet supported by Hermes[6]. For instance, in its most recent release, Hermes still does not support for features such as Proxy, Promise, and Async calls. For this reason—low coverage on Test262—, we opted not to include Hermes with other JavaScript engines. Since we only

[6] See `https://github.com/facebook/hermes/blob/master/doc/Features.md`

fuzz test files that pass in all engines, this would require us to significantly reduce the number of tests used in the experiments, affecting overall results.

We followed the same methodology described in previous sections to evaluate Hermes. When experimenting with test transplantation, we noted many errors when replaying the test suite of SpiderMonkey on Hermes. This happened because SpiderMonkey has its own assertion framework, but Hermes was unable to interpret those assertions (because of missing features), resulting in failures in all test executions from. These failures did not happen when the SpiderMonkey tests were transplanted to the other engines, though. Overall, we reported three issues to the Hermes bug tracker as result of using test transplantation[7]. One of them was confirmed by the Hermes maintainers by the time of this submission. For another, the maintainer explained the issue was related to a feature not yet supported—a `return` statement outside the scope of a block. For the third issue, developers disagreed on the proper implementation of one particular part of the specification involved in the issue (i.e., whether a numeric escape character (non-octal-eight, `\8`) should be allowed or not in strict mode. The issue was closed, but the maintainers opened a new one to discuss this matter. By following the discussion, it was interesting to observe that maintainers are aware that this feature is implemented by other engines such as V8, ChakraCore, and SpiderMonkey. Yet, they were unsure how they should treat it in Hermes. Considering differential testing, we found and reported one bug on Hermes using radamsa. As of this writing, this particular bug report was not yet addressed by any Hermes maintainer.

Overall, we noted that test transplantation and differential testing were effective techniques in revealing potential bugs in non-trivial JavaScript engines of varying degrees of maturity. We found bugs both in extremely robust and largely adopted engines and also in recently developed engines. The issue diagnosis of developers seem to differ though. Issues reported to stable engines are more likely to be real bugs—as lots of problems have been already scrutinized—whereas issues reported to new engines are more likely to be inconsistencies related to influx development as opposed to real bugs.

8 Discussion

This section discusses bug reports, threats to validity, and key findings and lessons learned.

8.1 Bug Reports

We issued several bug reports as result of this work. For space, we are unable to discuss all of them. We sampled some bug reports to discuss in the following. The selection criteria we used was:

[7] `https://github.com/facebook/hermes/issues/<id>`, with id 265, 266, 267.

1. to cover all engines we found bugs–ChakraCore, Hermes, JSC, SpiderMonkey, V8 (see Figure 1);
2. to cover each technique–test transplantation and differential testing;
3. to cover a case of rejected bug report;
4. to use short tests (for space).

Issue #19, Table 6. The code snippet below shows the test input we used to reveal a bug in **JSC** version 234555.

```
var obj = {}; var arr = [];
try { arr.sort(obj); assert(false);}
catch (e) { assert(e instanceof TypeError); }
```

This is a test case of the JerryScript suite. The bug was found during the test transplantation experiment. According to the EcmaScript spec [76], the parameter to the `Array.sort` function should be a comparable object or an undefined value, otherwise it should throw a `TypeError`. In this case, JSC incorrectly accepts a non-callable object as argument to `sort` and the test fails in the subsequent step. The other engines raise a `TypeError` as expected.

Issue #32, Table 6. We reported the snippet `"use strict" 010` to the **SpiderMonkey** development team as a bug and the bug was confirmed in a few hours. This is a test case originally from the Hermes suite using test transplantation. According the specification, in strict mode, the engine must use the prefix '0o' or '0O' to represent octal numeric literals. This is an issue that explores a deprecated octal token after ASI (Automatic Semicolon Insertion). The other engines throw a SyntaxError due to the missing of the octal token, but SpiderMonkey returns an integer `8` that represents `010` in octal.

Issue #2, Table 9. We reported the code snippet below to the **ChakraCore** development team as a bug, but they did *not* accept.

```
function test() {
  return typeof String.prototype.repeat === "function"
    && "foo".repeat(657604378) === "foofoofoo"; }
```

This is a test case original from the JSC suite that the radamsa fuzzer modified. The original test used the integer literal 3 as argument to `repeat()`, i.e., the expression produced a string with three repetitions of the string "foo". The new test uses a long integer instead as parameter to `repeat()`. As result, the engine crashes. The team answered that this was an incompatibility by design[8] as the function was not expected to receive such a long value.

Issue #4, Table 9. The code snippet below shows a test that reveals a bug in **ChakraCore**.

[8] We interpreted as a violation of an undocumented precondition

```
{ var a = {valueOf: function(){ return "\x00"}}
  assert(+a === 0) }
```

The object property `valueOf` stores a function that returns a primitive value identifying the target object [6]. The original version of this code returns an empty string whereas the version of the code modified by the radamsa fuzzer [34] returns a string representation of a null character (`NUL`). The unary plus expression "`+a`", used in the assertion, is equivalent to the operation `ToNumber(a.valueOf())` that converts a string to a number, otherwise the operation returns NaN (Not a Number) [7]. This test fails in all engines but ChakraCore. For all three engines the string cannot be parsed as an hexadecimal. As such, they produce a NaN and the test fails as expected. ChakraCore, instead, incorrectly converts the string to zero, and the test passes. As Table 9 shows, the ChakraCore team fixed the issue soon after our report.

Issue #18, Table 9. The snippet `eval(function b(a){break;});` revealed a bug in **V8** version 7.0.244. This code snippet was obtained by fuzzing a Test262 test with quickfuzz. In its original version, a string (omitted for space), passed as argument to the `eval` function, encoded the actual test. The fuzzer replaced the string argument with a function whose body is a `break` statement outside a valid block statement. Section B.3.3.3 from the EcmaScript spec [24] documents how eval should handle code containing function declarations. According to the spec [27], the virtual machine should throw an early error–in this case, a SyntaxError–if the break statement is not nested in a loop or switch statement. All engines, but V8, behave as expected in this case.

Bug reported on Hermes. The snippet `b+/v/a` represents a bug confirmed in **Hermes** engine with differential testing using radamsa. The original test case contains a string concatenation. This is a case of validation of RegEx flags. The fuzzer mutates the file with a regular expression `/v/` after the plus operator. In this case, the plus operator turns into a flag of the RegEx `+/v/`, but this flag is not valid. The expected behavior is to throw an early SyntaxError due to the invalid regular expression flag, but Hermes seems not to treat the early validation of the regexp flags as explained by one of their developers.

8.2 Threats To Validity

As it is the case of most empirical evaluations, our findings are subject to internal, external, and construct threats to validity. Considering internal validity, conceptually, it is possible that the authors of this paper made mistakes in the implementation of the scripts supporting the experiments. To mitigate this threat, we carefully inspected the implementation and results, looking for inconsistencies whenever possible. As for external validity, our results might not generalize to other test inputs and engines. We though carefully selected inputs from various sources according to a well-defined criteria (Section 4). Likewise, we selected the engines by using using a documented criteria (Section 3) and found that the engines selected were associated, certainly not by

coincidence, with the browsers informally considered the most popular in the market. A reader might argue that since we mined files from other JS engines, such as Duktape, JerryScript, JSI, and Tiny-js, we could also have ran test transplantation on them. We decided not to do so for different reasons: JSI was deprecated, TinyJS is not actively maintained. Finally, at the time we started this work, Duktape and JerryScript did not provide support to ES6. We checked this information again in Jun 2020, and they are still only partially supporting ES6. In terms of construct validity, we used standard metrics to determine the effectiveness of the testing techniques we studied (e.g., number of bugs confirmed and fixed and severity). Engine developers were responsible for determining the labels of the bug reports and their severity. Consequently, these metrics originate from a trusted source.

8.3 Key Findings

The main findings of this study are as follows.

1. Both techniques we studied have shown to be practical and effective to find bugs on real, complex, and widely used software systems;
2. Even for language APIs with fairly clear specs, as it is the case of JavaScript, there is likely (a lot of) variation between different implementations, which brings intrinsic challenges to developers that work on them;
3. Even simple black-box fuzzers can create surprisingly interesting inputs;

The main findings of this study are as follows: 1) The techniques we selected found, with relatively low effort, several bugs even in very robust engines, such as Mozilla's SpiderMonkey, 2) Even for software projects with fairly clear specifications, as the case of JavaScript [78], there are lots of undefined implementation-specific behaviors whose implementation can inadvertently lead to interference in the well-specified parts of the spec, leading to clear bugs. 3) Finding functional/non-crash bugs with differential testing is feasible on real, complex, widely used pieces of software. Even black-box mutational fuzzers revealed bugs. We do expected that other kinds of fuzzers (e.g., gray-box fuzzers and black-box grammar-based generational fuzzers) could reveal even more bugs

8.4 Key Lessons

The key lessons of this study are as follows:

1. The use of Mozilla's SpiderMonkey and Google's V8 engines should be encouraged;
2. The cost of inspection of the warnings in differential testing should be reduced;

3. Finding bugs in large software is a very effective way to engage students in contributing to open-source software and learning Software Testing practices.

1) When taking into consideration our bug finding campaign, Mozilla's SpiderMonkey and Google's V8 stood out as the most reliable engines. Therefore, we recommend the use of these engine for running JS applications. 2) Further reducing cost of inspection in differential testing is an important problem. Although the inspection activity was not uninterrupted, it is safe to say that each warning required a substantial amount of time to analyze for potential false alarms. In fact, many hi warnings reported with differential testing were not analyzed. We observed empirically that the cost of analysis was proportional to (i) the JS specification covered by the original test (as developers need to read and understand those parts) and (ii) the availability of alternative implementations. We prefer to see such problem as an opportunity for future research. For example, applying learning techniques to prioritize the warnings more likely to be faulty (in the spirit of the work of Chen and colleagues [16]) may be a promising avenue to explore. Recall that the rate of true positives of the techniques we studied is rather small. 3) We learned that reporting real bugs is a great way to train (and encourage) students in software testing. Students praised the experience of diagnosing failures, understanding part of the specs (as needed), writing bug reports, participating in discussions on issue trackers, and observing the change of status. That was a relatively self-contained hands-on activity that enabled students to engage in a real-life serious industrial project.

9 Related Work

9.1 Differential Testing

Several different applications of differential testing have been proposed in recent years. Chen and colleagues [15] recently proposed a technique to generate X.509 certificates based on Request For Proposals (RFC) as specification with the goal of detecting bugs in different SSL/TLS implementations. Those bugs can compromise security of servers which rely on these certificates to properly authenticate the parties involved in a communication session. Lidbury and colleagues [49] and Donaldson and colleagues [22] have been focusing on finding bugs in programs for graphic cards (e.g., OpenCL). These programs use the Single-Instruction Multiple-Data (SIMD) programming abstraction and typically run on GPUs. Perhaps the application of differential testing that received most attention to date was compiler testing. In 1972, Purdom [66] proposed the use of a generator of sentences from grammars to test correctness of automatically generated parsers. After that, significant progress has been made. Lammel and Shulte proposed Geno to cross-check XPath implementations using grammar-based testing with controllable combinatorial coverage [46]. Yang and colleagues [86] proposed CSmith to randomly create C programs from a

grammar, for a subset of C, and then check the output of these programs in different compilers (e.g., GCC and LLVM). Le and colleagues [47] proposed "equivalence modulo inputs", which creates variants of program which should have equivalent behavior compared to the original, but for which the compiler manifests discrepancy. Differential testing has also been applied to test refactoring engines [21], to test symbolic engine implementations [43], to test disassemblers and binary lifters [62,44], and very recently to test JavaScript debuggers [48]. All in all, it has shown to be flexible and effective for a wide range of applications. Surprisingly, not much work has been done on differential testing of JS engines. Mozilla uses differential testing to look for discrepancies across different configurations of the same version of its SpiderMonkey engine (using the "compare_jit" flag of jsfunfuzz [57]) whereas we focus on discrepancy across engines. Patra and Pradel evaluated their language-agnostic fuzzing strategy using differential testing. Their focuses on finding differential bugs across multiple browsers [64]. As such they specialized their fuzzer to HTML and JS (see Section 9.3). In contrast to Patra and Pradel, we did not propose new techniques; our contribution was empirical.

9.2 Testing JS Programs

Patra and colleagues [63] proposed a lightweight approach to detect conflicts in JS libraries that occur when names introduced by different libraries collide. This problem was found to be common as the design of JS allows for overlaps in namespaces. A similar problem has been investigated by Nguyen and colleagues [61] and Eshkevari and colleagues [29] in the context of PHP programs, which are popular in the context of Content Management Systems as WordPress. The focus of this paper is on testing JS engines as opposed to JS programs. Our goal is therefore orthogonal to theirs.

9.3 Testing JS Engines

The closest work to ours was done by Patra and Pradel [64]. Their work proposes a language-agnostic fuzzer to find cross-browser HTML+JS discrepancies. The sensible parts of the infrastructure they built are the checks of input validity (as to reduce waste/cost) and output correctness (as to reduce false positives). Patra and Pradel work is complementary to ours–in principle, we could use their fuzzer in our evaluation. The main difference of our work to theirs is in goal–we aim at assessing reliability of JS engines and find bugs on them using simple approaches whereas they aim at proposing a new technique.

Fuzzing is an active area of investigation with development of new techniques both in academia and industry. Several fuzzing tools exist focused on JS. Section 5.3 briefly explain different fuzzing strategies and tools. Existing techniques prioritize automation with a focus on finding crashes; see the sanitizers used in libFuzzer [31], for instance. In general, it is important for these

tools that a warning reveals something potentially alarming as a crash given that fuzzing is a time-consuming operation, i.e., the ratio of bugs found per inputs generated is often very low. Our approach contrasts with that aim as we focus on finding errors manifested on the output, which rarely result in crashes and, consequently, would go undetected by current fuzzing approaches. It is should be noted, however, that such problems are not unimportant as per the severity levels reported in Tables 6 and 9.

10 Conclusions

JavaScript (JS) is very popular today. Bugs in engine implementations often affect lots of people and organizations. Implementing correct engines is challenging because the specification is intentionally incomplete and evolves frequently. Finding bugs in JS engines is challenging for similar reasons.

This paper reports on a study to evaluate two techniques for finding bugs in JS–test transplantation and cross-engine differential testing. The first technique runs the test suite of one given engine in another engine. The second technique fuzzes existing inputs and then compares the output produced by different engines with a differential oracle.

We found that both techniques were very effective at finding bugs in JS engines. Overall, we reported 59 bugs in this study. Of which, 39 were confirmed by developers and 29 were fixed. Although more work is necessary to reduce cost of manual analysis, we found that our results provide strong evidence that exploring test transplantation and differential testing should be encouraged to find functional bugs in JavaScript engines.

In the near future, we plan to explore techniques to prioritize the warnings reported by these techniques and to continue involving students in the task of diagnosing these warnings. Using learning techniques for prioritization [56], similar to what Chen and colleagues [16] did to prioritize the warnings reported by CSmith [86], seems a promising starting point for this improvement.

The scripts to run the experiments for this study will be available upon request. The data is publicly available `https://github.com/damorimRG/entente/`.

Acknowledgments. Igor is supported by the FACEPE fellowship IBPG-0123-1.03/17. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

References

1. American fuzz loop. `https://web.archive.org/web/20200530065331/http://lcamtuf.coredump.cx/afl/`
2. Getting started with libfuzzer at chromium. `https://web.archive.org/web/20200530065331/https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/getting_started.md`
3. Libfuzzer. `https://web.archive.org/web/20200530065331/https://llvm.org/docs/LibFuzzer.html`

4. libfuzzer tutorial. https://web.archive.org/web/20200530065331/https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md
5. Lithium. https://web.archive.org/web/20200530065331/https://github.com/MozillaSecurity/lithium
6. Object.valueof documentation. https://web.archive.org/web/20200530065331/https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/ValueOf
7. Unary plus - es6 specifications. https://web.archive.org/web/20200530065331/https://www.ecma-international.org/ecma-262/8.0/index.html#sec-unary-plus-operator
8. Apple: Severity levels WebKit bugs (JavaScriptCore). https://web.archive.org/web/20200530065331/https://webkit.org/bug-prioritization/ (2018)
9. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A.: Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pp. 1690–1701. ACM, New York, NY, USA (2016). DOI 10.1145/2976749.2978383. URL http://doi.acm.org/10.1145/2976749.2978383
10. Babel: Babel Project. https://web.archive.org/web/20200530065331/https://github.com/babel/babel
11. BlogEngine.Net: BlogEngine.Net Project. https://web.archive.org/web/20200530065331/https://github.com/rxtur/BlogEngine.NET
12. Brian Terlson: escli-host. https://web.archive.org/web/20200530065331/https://github.com/bterlson/eshost-cli
13. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS'07, pp. 15:1–15:16. USENIX Association, Berkeley, CA, USA (2007). URL http://dl.acm.org/citation.cfm?id=1362903.1362918
14. Camilo Bruni–V8 engineer: for-in undefined behavior. https://web.archive.org/web/20200530065331/https://v8project.blogspot.com/2017/03/fast-for-in-in-v8.html (2018)
15. Chen, C., Tian, C., Duan, Z., Zhao, L.: Rfc-directed differential testing of certificate validation in ssl/tls implementations. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 859–870. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180226. URL http://doi.acm.org/10.1145/3180155.3180226
16. Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., Xie, B.: Learning to prioritize test programs for compiler testing. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 700–711. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/ICSE.2017.70. URL https://doi.org/10.1109/ICSE.2017.70
17. Chen, Y., Su, T., Sun, C., Su, Z., Zhao, J.: Coverage-directed differential testing of jvm implementations. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, pp. 85–99. ACM, New York, NY, USA (2016). DOI 10.1145/2908080.2908095. URL http://doi.acm.org/10.1145/2908080.2908095
18. Chen, Y., Su, Z.: Guided differential testing of certificate validation in ssl/tls implementations. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 793–804. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786835. URL http://doi.acm.org/10.1145/2786805.2786835
19. Chromium: Issue 4247. https://web.archive.org/web/20200530065331/https://bit.ly/2O08uW2
20. Chromium: V8 JavaScript Engine. https://web.archive.org/web/20200530065331/https://chromium.googlesource.com/v8/v8.git
21. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 185–194. ACM, New York, NY, USA (2007). DOI 10.1145/1287624.1287651. URL http://doi.acm.org/10.1145/1287624.1287651

22. Donaldson, A.F., Evrard, H., Lascu, A., Thomson, P.: Automated testing of graphics shader compilers. Proc. ACM Program. Lang. **1**(OOPSLA), 93:1–93:29 (2017). DOI 10.1145/3133917. URL http://doi.acm.org/10.1145/3133917
23. Duktape: Duktape. https://web.archive.org/web/20200530065331/https://github.com/svaarala/duktape
24. Ecma Internacional: Changes to EvalDeclarationInstantiation. https://www.ecma-international.org/ecma-262/8.0/index.html#sec-web-compat-evaldeclarationinstantiation
25. Ecma Internacional: Ecma Internacional. https://www.ecma-international.org
26. Ecma Internacional: Number.toPrecision specification. https://www.ecma-international.org/ecma-262/8.0/index.html#sec-number.prototype.toprecision
27. Ecma Internacional: Static Semantics: Early Errors. https://www.ecma-international.org/ecma-262/8.0/index.html#sec-break-statement-static-semantics-early-errors
28. ES: Array new attributed caused bugs. https://web.archive.org/web/20200530065331/https://esdiscuss.org/topic/array-prototype-values-is-not-web-compat-even-with-unscopables (2014)
29. Eshkevari, L., Antoniol, G., Cordy, J.R., Di Penta, M.: Identifying and locating interference issues in php applications: The case of wordpress. In: Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014, pp. 157–167. ACM, New York, NY, USA (2014). DOI 10.1145/2597008.2597153. URL http://doi.acm.org/10.1145/2597008.2597153
30. Facebook: React Native. https://web.archive.org/web/20200530065331/https://reactnative.dev/docs/hermes
31. Google: libFuzzer Tutorial. https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md
32. Google Chrome Lab: JSVU–JavaScript (engine) Version Updater. https://web.archive.org/web/20200530065331/https://github.com/GoogleChromeLabs/jsvu
33. Grieco, G., Ceresa, M., Buiras, P.: Quickfuzz: an automatic random fuzzer for common file formats. In: Proceedings of the 9th International Symposium on Haskell, pp. 13–20. ACM (2016)
34. Helin, A.: Radamsa fuzzer. https://web.archive.org/web/20200530065331/https://github.com/aoh/radamsa
35. Hermes: Hermes Project. https://web.archive.org/web/20200530065331/https://github.com/facebook/hermes/
36. Hodovan, R.: Grammarinator. https://web.archive.org/web/20200530065331/https://github.com/renatahodovan/grammarinator
37. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security'12, pp. 38–38. USENIX Association, Berkeley, CA, USA (2012). URL http://dl.acm.org/citation.cfm?id=2362793.2362831
38. JerryScript: JerryScript. https://web.archive.org/web/20200530065331/https://github.com/technosaurus/jsish
39. JerryScript: JerryScript Project. https://web.archive.org/web/20200530065331/https://github.com/jerryscript-project/jerryscript
40. Joe Belfiore: Microsoft Edge: Making the web better through more open source collaboration. https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/#IIycUFupVTBcbAY5.97 (2018)
41. John Resig: JavaScript in Chrome. https://web.archive.org/web/20200530065331/https://johnresig.com/blog/javascript-in-chrome/ (2018)
42. Kangax: Ecmascript6 compatibility. https://web.archive.org/web/20200530065331/http://kangax.github.io/compat-table/es6/

43. Kapus, T., Cadar, C.: Automatic testing of symbolic execution engines via program generation and differential testing. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 590–600. IEEE Press, Piscataway, NJ, USA (2017). URL http://dl.acm.org/citation.cfm?id=3155562.3155636
44. Kim, S., Faerevaag, M., Jung, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 353–364. IEEE Press, Piscataway, NJ, USA (2017). URL http://dl.acm.org/citation.cfm?id=3155562.3155609
45. Kusner, M., Sun, Y., Kolkin, N., Weinberger, K.: From word embeddings to document distances. In: International Conference on Machine Learning, pp. 957–966 (2015)
46. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: M.Ü. Uyar, A.Y. Duale, M.A. Fecko (eds.) Testing of Communicating Systems, pp. 19–38. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
47. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp. 216–226. ACM, New York, NY, USA (2014). DOI 10.1145/2594291.2594334. URL http://doi.acm.org/10.1145/2594291.2594334
48. Lehmann, D., Pradel, M.: Feedback-directed differential testing of interactive debuggers. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pp. 610–620 (2018). DOI 10.1145/3236024.3236037. URL https://doi.org/10.1145/3236024.3236037
49. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, pp. 65–76. ACM, New York, NY, USA (2015). DOI 10.1145/2737924.2737986. URL http://doi.acm.org/10.1145/2737924.2737986
50. LLVM: clang documentation. https://web.archive.org/web/20200530065331/http://clang.llvm.org/docs/
51. 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 pp. 1–1 (2019)
52. Microsoft: ChakraCore. https://web.archive.org/web/20200530065331/https://github.com/Microsoft/ChakraCore
53. Microsoft: Severity levels chakra bugs. https://web.archive.org/web/20200530065331/https://github.com/Microsoft/ChakraCore/wiki/Label-Glossary (2018)
54. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013, pp. 3111–3119. Curran Associates Inc., Lake Tahoe, Nevada, USA (2013)
55. Miller, B.P.: Fuzz testing. https://web.archive.org/web/20200530065331/http://pages.cs.wisc.edu/~bart/fuzz/
56. Miranda, B., Lima, I., Legunsen, O., d'Amorim, M.: Prioritizing runtime verification violations. In: Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST) (2020). To appear
57. Mozilla: jsfunfuzz. https://web.archive.org/web/20200530065331/https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz
58. Mozilla: jsfunfuzz at Mozilla. https://web.archive.org/web/20200530065331/https://mzl.la/2LsctZL
59. Mozilla: SpiderMonkey Project. https://web.archive.org/web/20200530065331/https://github.com/mozilla/gecko-dev
60. Mozilla: Triage process for firefox components in mozilla-central and bugzilla. https://web.archive.org/web/20200530065331/https://github.com/mozilla/bug-handling/blob/master/policy/triage-bugzilla.md (2018)
61. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. In: ICSE, pp. 907–918 (2014)

62. Paleari, R., Martignoni, L., Fresi Roglia, G., Bruschi, D.: N-version disassembly: Differential testing of x86 disassemblers. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pp. 265–274. ACM, New York, NY, USA (2010). DOI 10.1145/1831708.1831741. URL http://doi.acm.org/10.1145/1831708.1831741
63. Patra, J., Dixit, P.N., Pradel, M.: Conflictjs: Finding and understanding conflicts between javascript libraries. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 741–751. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180184. URL http://doi.acm.org/10.1145/3180155.3180184
64. Patra, J., Pradel, M.: Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Tech. rep., TU Darmstadt, Department of Computer Science (2016)
65. Petsios, T., Tang, A., Stolfo, S., Keromytis, A.D., Jana, S.: Nezha: Efficient domain-independent differential testing. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 615–632 (2017). DOI 10.1109/SP.2017.27
66. Purdom, P.: A sentence generator for testing parsers. BIT Numerical Mathematics **12**(3), 366–375 (1972). DOI 10.1007/BF01932308. URL https://doi.org/10.1007/BF01932308
67. RedMonk: The RedMonk Programming Language Rankings: June 2018. https://web.archive.org/web/20200530065331/https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/ (2018)
68. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chap. Learning Internal Representations by Error Propagation, pp. 318–362. MIT Press, Cambridge, MA, USA (1986). URL http://dl.acm.org/citation.cfm?id=104279.104293
69. Simply Technologies: Why is JavaScript So Popular? https://web.archive.org/web/20200530065331/https://www.simplytechnologies.net/blog/2018/4/11/why-is-javascript-so-popular (2018)
70. de Ciencias de la Información y de Sistemas, C.I.F.A.: Quickfuzz. https://web.archive.org/web/20200530065331/http://quickfuzz.org/
71. Sivakorn, S., Argyros, G., Pei, K., Keromytis, A.D., Jana, S.: Hvlearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, pp. 521–538 (2017). DOI 10.1109/SP.2017.46. URL https://doi.org/10.1109/SP.2017.46
72. Stackify: Most popular and influential programming languages of 2018. https://web.archive.org/web/20200530065331/https://stackify.com/popular-programming-languages-2018/
73. StackOverflow community: Elements order in a "for (... in ...)" loop. https://web.archive.org/web/20200530065331/https://stackoverflow.com/questions/280713/elements-order-in-a-for-in-loop (2018)
74. TC39: Official ECMA262 Conformance Test Suite. https://web.archive.org/web/20200530065331/https://github.com/tc39/test262
75. TC39: TC39 GitHub repo. https://web.archive.org/web/20200530065331/http://tc39.github.io/
76. TC39: Array sort. https://web.archive.org/web/20200530065331/https://tc39.github.io/ecma262/\#sec-array.prototype.sort (2018)
77. TC39: ECMA262 repository. https://web.archive.org/web/20200530065331/https://tc39.github.io/ecma262/ (2018)
78. TC39: ECMA262 Spec. https://web.archive.org/web/20200530065331/https://www.ecma-international.org/ecma-262/8.0/ (2018)
79. TC39: TypeConversion, ToIndex function. https://web.archive.org/web/20200530065331/https://tc39.github.io/ecma262/#sec-toindex (2018)
80. The Chromium Project: Chromium bug labels. https://web.archive.org/web/20200530065331/https://www.chromium.org/for-testers/bug-reporting-guidelines/chromium-bug-labels (2018)
81. The Node.js Foundation: Node.js. https://web.archive.org/web/20200530065331/https://nodejs.org

82. Tiny-js: Tiny-js. https://web.archive.org/web/20200530065331/https://github.com/gfwilliams/tiny-js
83. Unknown: Mozilla. https://web.archive.org/web/20200530065331/https://github.com/mozilla/gecko-dev/tree/master/js/src/tests/non262
84. WebKit: WebKit Project. https://web.archive.org/web/20200530065331/https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore
85. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pp. 283–294. ACM, New York, NY, USA (2011). DOI 10.1145/1993498.1993532. URL http://doi.acm.org/10.1145/1993498.1993532
86. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pp. 283–294. ACM, New York, NY, USA (2011). DOI 10.1145/1993498.1993532. URL http://doi.acm.org/10.1145/1993498.1993532
87. Zhang, T., Kim, M.: Automated transplantation and differential testing for clones. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 665–676. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/ICSE.2017.67. URL https://doi.org/10.1109/ICSE.2017.67